An Efficient Embedded Software Development Approach for Multiprocessor System-on-Chips

Xavier Guerin

UNIVERSITÉ DE GRENOBLE

Numéro attribué par la bibliothèque
978-2-84813-154-2

THÈSE

pour obtenir le grade de **DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE**

Spécialité : « Informatique : Systèmes et Communication »

préparée au laboratoire TIMA dans le cadre de **l'École Doctorale « Mathématiques, Sciences et Technologies de l'Information, Informatique »**

préparée et soutenue publiquement par

Xavier Guérin

le 12 Mai 2010

Titre :

Approche Efficace de Développement de Logiciel Embarqué pour des Systèmes Multiprocesseurs sur Puce

---

sous la direction de Frédéric Pétrot et de Frédéric Rousseau

JURY

Pr. Jacques Mossière	Président
Pr. Gilles Muller	Rapporteur
Pr. Wolfgang Schröder-Preikschat	Rapporteur
Pr. Jean-luc Dekeyser	Examinateur
Dr. Pier Stanislao Paolucci	Examinateur
Pr. Frédéric Pétrot	Directeur
Pr. Frédéric Rousseau	Co-directeur

Remerciements

L'accomplissement d'une thèse de doctorat est rarement le fait d'une seule personne. Ainsi, j'aimerai, à l'occasion de ces quelques lignes, adresser mes remerciements à ceux qui ont participé à l'achèvement de ce travail. Aux professeurs Frédéric Pétrot et Frédéric Rousseau, pour leur confiance et toute la liberté qu'ils m'ont accordé dans la poursuite de mes recherches. À messieurs Olivier Muller, Nicolas Fournel et Quentin Meunier pour leurs conseils avisés et leur professionnalisme. À messieurs Pierre Guironnet-de-Massas, Guillaume Godet-Bar et Patrice Gerin pour leur constante collaboration et leur support amical. Au docteur Ahmed Amine Jerraya, qui, très tôt, a cru en mon projet et m'a permis de suivre la route qui est aujourd'hui la mienne. Enfin, à ma famille, sans qui rien n'aurait été possible.

Table des matières

Première partie

Résumé en Français

1
Introduction

Les systèmes embarqués ont considérablement évolué depuis leur première utilisation autour des années 1960, alors en l'état d'assemblages simples de transistors. Avec l'avènement et la généralisation du microprocesseur au milieu des années 1980, la plupart des opérations réalisées en matériel furent remplacées par leurs équivalents logiciels, interprétés par une unité d'exécution programmable. Parallèlement à cela, la nature même des applications embarquées a évolué.

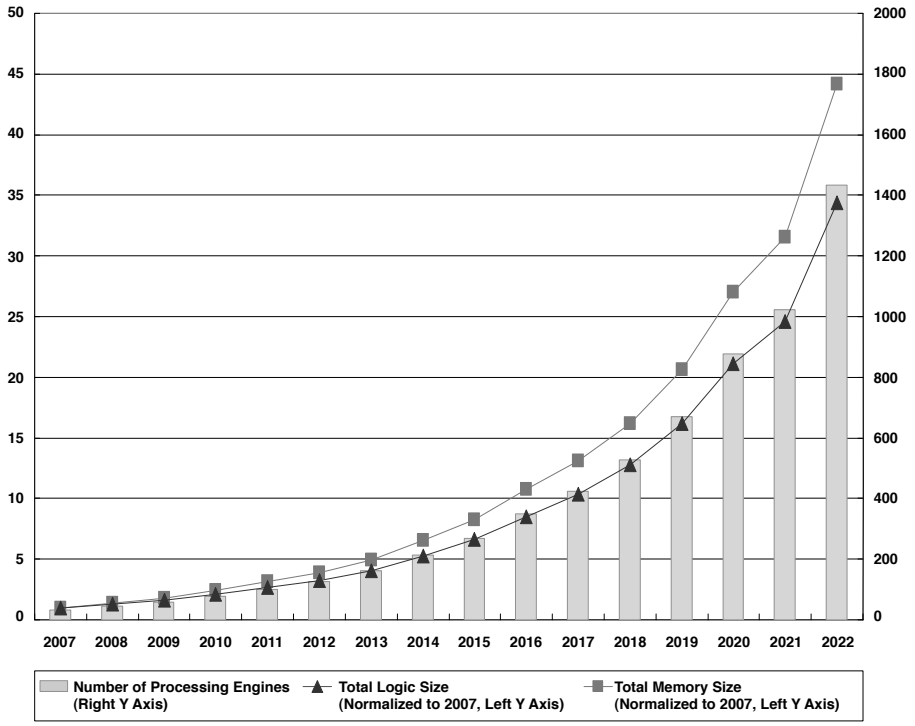


FIGURE 1.1 – Feuille de route technologique internationale pour les semi-conducteurs.

Les premières applications embarquées étaient en plusieurs points comparables aux applications conçues pour les ordinateurs du début des années 1980 : limitées en termes de fonctionnalités, construites *ex nihilo*, elles devaient faire face à de fortes contraintes matérielles

telles qu'une quantité de mémoire limité ou une fréquence de fonctionnement basse. De plus, elles devaient aussi compter sur les langages d'assemblage afin d'extraire du processeur la plus petite parcelle de performance. Néanmoins, les solutions modernes sont beaucoup plus élaborées.

D'un coté, les applications embarquées modernes contiennent des fonctionnalités très gourmandes en terme de puissance de calcul, comprenant des algorithmes de traitement de données complexes comme des décodeurs multimédia et/ou des protocoles de communication avancés. Elles sont aussi sujettes à des contraintes supplémentaires provenant de la nature de leurs plateformes cibles. La première contrainte concerne leur consommation électrique induite. La plupart des appareils embarqués sont alimentés par batterie. Ainsi, le programmeur doit ajouter certaines opérations au logiciel afin de réduire au minimum la consommation électrique de l'ensemble. La seconde contrainte est liée à la dynamique toute particulière du marché des systèmes embarqués. Afin de rester compétitif, les constructeurs se doivent de garder les coût et temps de production extrêmement bas tout en optimisant leurs temps de mise sur le marché (*time-to-market*, ou TTM).

De l'autre, les plateformes matérielles modernes sont plus efficaces. Suivant avec régularité la feuille de route technologique internationlae pour les semi-conducteurs (*international technology roadmap for semiconductors*, ou ITRS) (figure 1.1), le nombre de cœurs de calcul et la taille de la mémoire intégrés sur une seule puce augmentent constamment (avec la barrière psychologique des cent cœurs *on-chip* pronostiquée pour l'année 2010). En réalité, les constructeurs de matériels embarqués sont déjà à l'étude de solutions massivement multiprocesseurs (*massively multiprocessor* ou MMP), articulant plusieurs sous-systèmes processeurs identiques autour d'un réseau sur puce (*network-on-chip* ou NoC). Mais aujourd'hui, les concepteurs d'appareils embarqués comptent sur des architectures systèmes sur puce multiprocesseurs (*multiprocessor system-on-chip* ou MP-SoC), pour fournir le niveau de performance demandé par les applications.

1.1 L'architecture système sur puce

Un système sur puce (*system-on-chip* ou SoC) est une architecture matérielle qui intègre plusieurs composants électroniques sur un même circuit intégré. Ces composants sont reliés les uns aux autres par le biais d'une interconnexion *on-chip* telle qu'une matrice d'interconnexion, un bus ou un NoC. Parmi ces composant on peut trouver un microprocesseur ainsi qu'un certaine quantité de mémoire. Le microprocesseur peut-être un processeur à jeu d'instructions réduit (*reduced instruction set computer* ou RISC), un processeur de traitement de signal (*digital signal processor* ou DSP), ou bien encore un micro-contrôleur (uC). La quantité de mémoire *on-chip* est extrêmement petite ($\simeq$10Ko) et la latence d'accès aux données qu'elle contient est relativement courte (de l'ordre de un ou deux cycles d'horloge processeur).

Les autres composants sont typiquement des périphériques matériels tels que des contrôleurs d'entrées/sorties (*inputs/outputs* ou I/O), un contrôleur de mémoire externe, ou des accélérateurs matériels. Les premiers SoCs misaient fortement sur les accélérateurs matériels pour fournir aux applications logicielles un niveau de performance que le processeur seul n'aurait pas pu offrir. Comme les standards multimédias et les protocoles de communication évoluent rapidement, les concepteurs de propriétés intellectuelles (*intellectual property* ou IP), ne considèrent plus les approches purement matérielles comme viables. En conséquence, les accélérateurs matériels sont progressivement remplacés par plusieurs processeurs, moins chers à produire

1.1.1 Systèmes sur puce multiprocesseurs homogènes

Un système sur puce multiprocesseur homogène (*homogeneous* ou MP-SoC) est un SoC contenant plusieurs instances d'un même cœur de processeur généraliste. Ces cœurs sont accompagnés d'un petite quantité de mémoire *on-chip* appelée mémoire locale. L'ensemble {cœur(s), mémoire locale} est appelé sous-système cœur. La configuration MP-SoC homogène est en plusieurs points comparable à une architecture multiprocesseur symétrique (*symmetric multiprocessor* ou SMP) utilisée pour les machines de bureau ou les serveurs. Cette similarité fait sa plus grande force, car les approches systèmes existantes pour les architectures SMP peuvent aussi être utilisées sur des architectures MP-SoC homogènes. Néanmoins, leur ratio puissance/performance est plutôt médiocre tout particulièrement pour des applications nécessitant un calcul intensif, le domaine où les processeurs généralistes excellent étant le contrôle et non pas le calcul. L'introduction de cœurs de calcul spécifiques à un algorithme ou une application fournit une solution à ce problème.

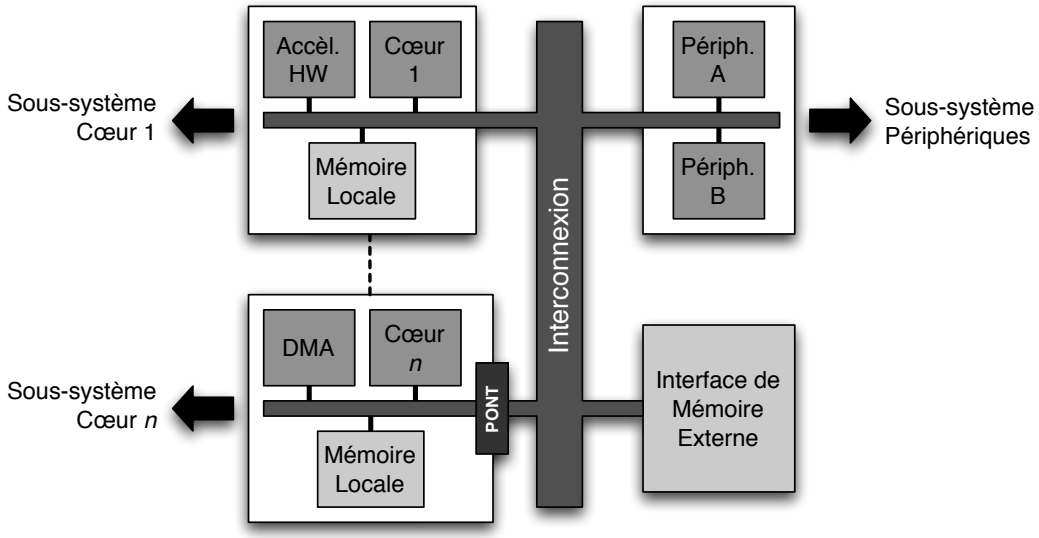


FIGURE 1.2 – Exemple de MP-SoC hétérogène.

1.1.2 MP-SoCs hétérogènes

Un MP-SoC hétérogène est un MP-SoC intégrant de multiples cœurs de processeurs non pas d'un, mais plusieurs types différents (figure 1.2). Ces cœurs peuvent être rangés en deux catégories : les processeurs généralistes (*general-purpose processor* ou GPP) excellant en contrôle avec des performances moyennes en calculs intensifs, et les processeurs spécifiques (*special-purpose processor* ou SPP) excellant dans leurs domaines de calculs avec des performances moyennes en contrôle. Au contraire des MP-SoCs homogènes qui intègrent une seule et unique mémoire locale pour tous les processeurs, chaque cœur hétérogène est accompagné de sa propre mémoire locale. Chaque sous-système cœur peut soit être directement connecté à l'interconnexion principale ou bien avoir son propre mécanisme de communication connecté à l'interconnexion principale par le biais d'un pont (*bridge*).

L'idée principale d'un tel concept matériel tient à la fois d'une stratégie *diviser pour régner* et du fait que les SoCs sont généralement dédiés à une classe d'applications donnée. En particulier, le concept de MP-SoC se concentre sur l'amélioration des performances d'une appli-

cation logicielle. Une application logicielle est la réalisation, utilisant un langage de programmation logicielle, d'un algorithme particulier combinant à la fois des opérations de contrôle de flot et des opérations de traitement de données. En utilisant des fréquence opérationnelles réduites, un niveau de performance équivalent peut être atteint lorsque ces opérations sont distribuées sur plusieurs cœurs de calcul spécialisés plutôt que sur plusieurs cœurs de calcul généralistes. De la même manière, cette solution favorise la réduction des consommation électrique, dissipation thermique, et divers coûts de production de l'ensemble.

Mais un type d'architecture aussi performant ne vient pas sans un certain coût : les MP-SoC hétérogènes sont extrêmement difficiles à programmer efficacement. Un programmeur développant du logiciel pour une telle architecture devra faire face à différents jeux d'instructions, différentes fréquences opérationnelles, différentes types et tailles de données, ainsi qu'à des accès mémoires non-uniformes — le logiciel peut ne pas avoir la même vision de la carte mémoire du matériel en fonction du processeur qui l'exécute. De plus, chaque cœur peut avoir un mécanisme qui lui est propre pour accéder à des données situées en-dehors de son sous-système. Non seulement ces différences rendent la programmation individuelle de chaque cœur plus difficile, mais elles créent aussi des problèmes de communication et de synchronisation lorsque deux programmes s'exécutant sur deux processeurs hétérogènes doivent s'échanger des données.

1.2 Objectif de ce travail

La complexité des architectures MP-SoC rend les méthodes de programmation traditionnelles obsolètes. Aucune application ne peut être entièrement développée à la main en temps raisonnable tout en tirant partie efficacement de telles plateformes. Seule une méthode qui a) prend en compte en même temps toutes les subtilités et capacités de l'architecture et b) offre un environnement de programmation avancé qui abstrait la complexité du matériel et automatise la production du code peut atteindre des niveaux d'efficacité acceptables. L'objectif de ce travail est de proposer une telle méthode.

1.3 Organisation de ce résumé

Ce résumé est structuré comme suit. Le chapitre 2 présente une brève analyse des techniques industrielles de développement de logiciel pour plateformes MP-SoC. Le chapitre 3 montre comment l'adjonction de mécanismes orientés composants et objets à un flot de conception existant permet d'améliorer le processus de développement d'applications embarquées. Le chapitre 4 donne une rapide description de l'environnement APES[1] et du micro-noyau DNA-OS[2]. Une conclusion à ce résumé est donnée en chapitre 5.

[1] Application Elements for System-on-Chip (SoC).
[2] DNA's Not just Another Operating System.

2
MP-SoC : un défi de programmation

Ce chapitre présente une analyse des principales approches utilisées dans l'industrie pour le développement de logiciels embarqués sur des plateformes MP-SoC. Cette analyse s'articule autour de quatre critères fondamentaux : flexibilité, capacité de mise à l'échelle, portabilité et automatisation.

Flexibilité : Plus une application devient complexe, plus les composants logiciels ont des chances d'être réutilisés. De la même manière, les développeurs de logiciel favorise généralement la production et la manipulation de codes binaires légers ne contenant que le strict nécessaire, plutôt que de codes binaires surchargés de fonctions inutiles et difficilement compréhensibles.

Mise à l'échelle : D'une architecture à l'autre, le nombre de processeurs disponibles peut changer et un périphérique matériel peut avoir été ajouté ou bien enlevé. La migration d'une application de l'ancienne architecture vers la nouvelle ne devrait être qu'une question de configuration et non de reprogrammation.

Portabilité : Si une fonction n'est pas radicalement dépendante du matériel, elle ne devrait pas être réalisée autant de fois qu'il existe de plateformes cibles différentes. La corollaire à cette constatation est que le portage d'une application d'une plateforme vers une autre devrait être rapide et sans douleur.

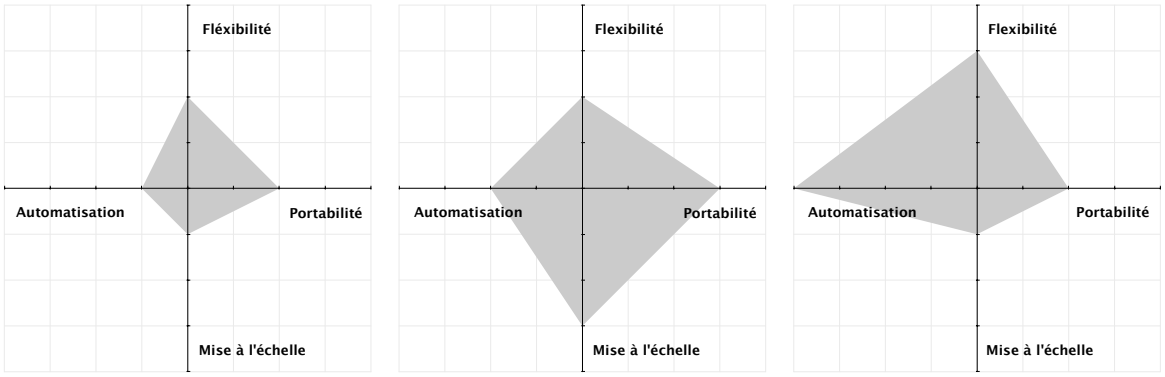
Automatisation : Il se peut qu'une simple application requiert plusieurs services systèmes différents, des bibliothèques externes, des pilotes de périphériques, etc. La sélection et la spécialisation de ces différents éléments devrait être automatique. De plus, un développeur peut exprimer le désir d'utiliser un niveau d'abstraction plus haut que le simple code C pour décrire son application. Ainsi, le code source correspondant à son application devrait être généré automatiquement et compatible avec la plateforme cible.

Le critère de flexibilité reflète la capacité d'une approche à favoriser la réutilisation de codes existants ainsi qu'à optimiser la taille de l'objet binaire produit. Le critère de mise à l'échelle exprime la capacité d'une approche à augmenter ou bien réduire ses fonctionnalités en rapport à une architecture donnée (e.g. nombre de processeurs, une ou plusieurs puces, etc.). La critère de portabilité évalue le coût associé à l'adaptation d'une approche donnée à une ou plusieurs plateformes matérielles différentes. Le critère d'automatisation indique la capacité

d'une approche à réduire les paramètres de temps et de coûts de ses cycles de développement logiciels en utilisant des processus automatisés.

2.1 Approche autonome

L'approche de développement autonome est l'approche la plus directe pour débuter le développement d'une application logicielle sur une plateforme embarquée. Ainsi dans sa forme la plus simple ne requiert-elle qu'un compilateur croisé (*cross-compiler*) même si les développeurs de logiciel ont généralement accès à différent paquets de support de carte (*board support package* ou BSP). Un BSP est une bibliothèque logicielle qui fournit des accès à chaque périphérique matériel présent sur une plateforme. Il peut être utilisé au travers de sa propre interface de programmation d'application (*application programming interface* ou API) et est spécifique à un matériel donnée. Contrairement à un système d'exploitation, les BSP ne fournissent aucun gestionnaire système. Ils sont généralement utilisés lorsqu'une solution plus complexe n'est ni nécessaire, ni financièrement abordable.



(a) Standalone (b) GPOS-based (c) Model-based

FIGURE 2.1 – Évaluation des principales approches de conception.

Cette approche de développement souffre de son manque de mise à l'échelle ainsi que de son manque d'automatisation (figure 2.1(a)). Même si elle fait plutôt bonne figure en termes de flexibilité et de portabilité, elle est clairement réservée à de petits projets au cycle de vie relativement long et qui requièrent un contrôle complet du matériel sur lequel ils s'exécutent.

2.2 Approche utilisant un système d'exploitation généraliste

Un système d'exploitation généraliste, ou GPOS, est un système d'exploitation complet conçu pour fournir un vaste panel de fonctionnalités à tous types d'applications. Parmi ces fonctionnalités on trouve généralement la gestion de plusieurs fils d'exécution (*multithreading*), la gestion de plusieurs processeurs identiques (*multiprocessing*), la protection de la mémoire et le support d'infrastructures réseaux. Les GPOS n'ont pas été spécifiquement conçu pour s'exécuter sur des plateformes MP-SoC. Ils sont le plus souvent simplement adaptés à partir de solutions existantes dans d'autres domaines de l'informatique comme les solutions de bureau ou les serveurs afin de fournir un environnement de développement comparable à

ceux auxquels les programmeurs sont généralement habitués. Ils sont utilisés lorsque la plateforme matérielle cible contient suffisamment de ressources (puissance du processeur, taille de la mémoire disponible) et lorsqu'une solution spécifique n'est pas particulièrement requise.

Cette approche de développement obtient de bons résultats en termes de portabilité et de mise à l'échelle grâce au très grands nombres d'architectures matérielles et de mécanismes logiciels supportés (figure 2.1(b)). Néanmoins, elle obtient des résultats médiocres en termes de flexibilité et d'automatisation. En effet, l'architecture monolithique des GPOS tend à produire des objets binaires gros, sous-optimisés et, même s'ils offrent des environnement de programmation typiques, l'approche de développement qui leur est associée se résume souvent à de la programmation manuelle. En conséquence, cette approche est le plus souvent utilisée pour des projets qui requièrent un solide support système et qui ne sont pas trop limités en terme de ressources matérielles.

2.3 Approche orientée modèles

La conception d'applications à base de modèles trouve sa source dans l'approche de conception au niveau système (*system-level design* ou SLD) où la réalisation d'une application est découplée de sa spécification. En lieu et place de langages de programmation standard, des modèles de calculs mathématiques formels sont utilisés pour décrire le comportement d'une application. Certaines dépendances logicielles, comme certaines fonctionnalités spécifiques aux systèmes d'exploitation temps-réel (*real-time operating system* ou RTOS), peuvent aussi être modélisées. Ceci permet au concepteur du logiciel d'accéder à un ensemble de simulations fonctionnelles rapides et, ainsi, de valider son application très tôt dans le processus de développement. Cette solution est généralement utilisée lorsque le comportement de l'application doit être consciencieusement validé et qu'une validation du logiciel rapide et précise est nécessaire.

Grâce à ses outils de génération de code et ses mécanismes de bibliothèque externe, le développement de logiciel à base de modèles obtient un très bon score en terme d'automatisation et un score honorable en terme de flexibilité, même si son incompatibilité avec des applications non conçues en utilisant un modèle de calcul supportés peut être problématique. Ses résultats en termes de mise à l'échelle et de portabilité sont relativement mauvais. Ceci n'est pas réellement surprenant étant donné que les approches automatisées se reposent généralement sur des modèles d'architectures bien maîtrisés afin de générer correctement le code applicatif. De plus, ce dernier étant garanti comme se comportant de la même manière que son modèle original, tout indéterminisme doit être évité. En conséquences, seules des architectures simples et uniprocesseurs sont généralement supportées. Ainsi, des projets critiques en termes de temps et de sécurité peuvent tirer partie d'une telle approche.

2.4 Analyse et contribution

Bien que chacune de ces approches amène sa part d'avantages — mise à part l'approche autonome, bien trop primitives pour être d'une réelle utilité — aucune d'elles ne satisfait nos besoins. L'approche basée sur un GPOS offre un ensemble de services de haut-niveau ainsi qu'un environnement de programmation bien connu du programmeur mais ne peut clairement pas tirer partie de configurations hétérogènes. Concernant les GPOS, aucun d'eux n'a

été conçu pour piloter des configurations multiprocesseurs hétérogènes et demanderaient un énorme travail de restructuration afin d'en être capable : tandis que cette opération est tout simplement impossible avec des GPOS commerciaux à sources fermées, cette tâche serait herculéenne quand bien même les sources du système seraient accessibles. Le support de l'hétérogénéité doit être inclue très tôt dans la conception d'un système d'exploitation. A cause des contraintes liées aux modèles de calcul sur lesquels elle se base, l'approche orientée modèles ne supporte qu'approximativement les plateformes multiprocesseurs homogènes, sans parler d'hétérogénéité. De plus, la plupart des solutions de génération de code sont parties intégrantes d'environnements commerciaux hors de prix, aux sources fermées et qui, en conséquence, ne peuvent être adapté aux besoins du matériel cible. De la même manière que pour l'approche utilisant un GPOS, l'intégration de l'hétérogénéité doit être faire très tôt lors de la conception du générateur.

Cette dissertation montre que des applications embarquées complexes peuvent tirer partie efficacement de plateformes MP-SoC hétérogènes tout en respectant les critères de flexibilité, mise à l'échelle, portabilité et temps de mise sur le marché. Elle fait la description d'un flot de conception de logiciel embarqué amélioré combinant un générateur de code, GECKO, et un environnement logiciel innovant, APES, afin d'obtenir un haut niveau d'efficacité. Notre contribution est double : 1) un flot de conception de logiciel embarqué amélioré avec un ensemble d'outils permettant la construction automatique d'objets binaires minimaux pour une application donnée ciblant une plateforme MP-SoC donnée (chapitre 3), et 2) un ensemble de composants logiciels modulaire et portable incluant des mécanismes de systèmes d'exploitations traditionnels ainsi que le support de multiples processeurs (chapitre 4).

3
Flot de conception de logiciel embarqué

Dans le chapitre précédent, nous avons vu qu'aucune des approches industrielles existantes ne satisfait simultanément les quatre critères fondamentaux — flexibilité, mise à l'échelle, portabilité et automatisation — alors qu'une approche optimale pour le développement de logiciel sur des plateformes MP-SoC devrait les réunir tous. Dans ce chapitre, nous présentons un flot de conception qui, en combinant à la fois un générateur de code applicatif et un mécanisme de sélection de composants logiciels, combine les critères de flexibilité et d'automatisation.

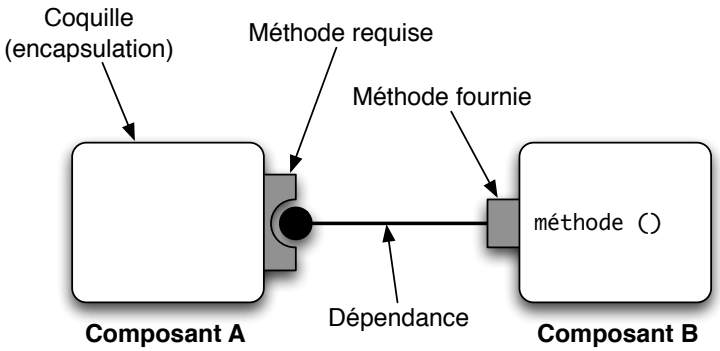


FIGURE 3.1 – Structure d'un composant logiciel.

Un des concepts clés de notre approche est le paradigme de conception à base de composant (*component-based design* ou CBD), définissant la notion de composant logiciel. Un composant logiciel est une organisation logicielle qui contient la réalisation de différentes fonctionnalités (voir figure 3.1). Elle combine une interface, qui exporte les méthodes que le composant fournit et requiert, avec une "coquille", qui contient les réalisations des méthodes fournies. Le lien entre une méthode fournie par un composant et requise par un autre composant et appelé dépendance.

Cette organisation est habituellement mise en place dans le cadre d'un contexte spécifique où les méthodes définies se réfèrent à des types ou des abstractions particulières afin de limiter les risques de recouvrement. Chacune des méthodes peut être soit publiquement accessible

(i.e appropriée pour l'utilisation par d'autres composants), soit privée au composant. Les méthodes publiques sont accessibles par le biais de leurs signatures. L'interface du composant contient l'union des signatures de ses méthodes publiques ainsi que l'union des signatures des méthodes utilisées dans sa réalisation. La présence des signatures requises par un composant est importante pour permettre la résolution de ses dépendances. De plus, l'interface d'un composant est, d'un point de vue extérieur à ce dernier, le seul et unique accès à ses méthodes. En conséquence, sa réalisation peut être complètement opaque vis-à-vis d'autres composants.

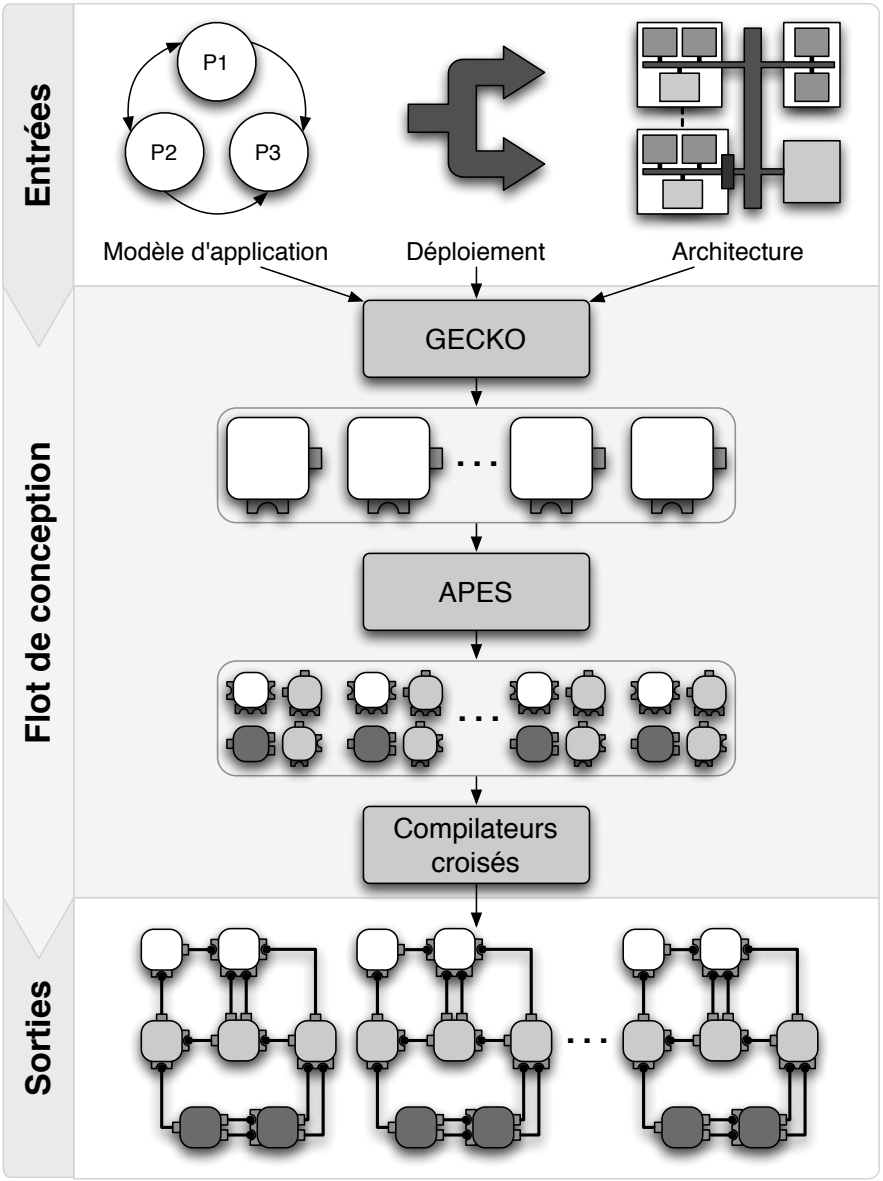


FIGURE 3.2 – Flot de conception de logiciel embarqué.

Cette déclaration est valide car aucune information concernant les mécanismes internes d'une réalisation n'est partagée, forçant ainsi les composants externes à se fier uniquement à la fonctionnalité explicite d'une méthode et non à ses effets de bord. Une fois définies, les interfaces doivent être déclarées afin d'être exploitables. Historiquement, les développeurs reposent sur un langage de description d'interfaces (*interface definition language* ou IDL),

pour effectuer cette tâche. Les langages de ce type décrivent l'interface d'un composant sans avoir connaissance du langage utilisé pour effectivement réaliser le comportement de cette interface [38, 48, 82]. Une fois les descriptions écrites, elles sont habituellement compilées en une application qui sera plus tard utilisée pour générer des spécifications d'interfaces compatibles avec le langage de programmation utilisé dans leurs réalisations. Non seulement ces interfaces améliorent la réutilisation de composants existants mais elles accélèrent aussi le processus de programmation : pour un composant donné, et avec l'aide de primitives comme *provide* ou *require*, un graphe de dépendances logicielles ainsi que son objet binaire correspondant peuvent être automatiquement générés.

Basée sur ce paradigme, l'approche de développement logicielle que nous proposons dans ce travail peut être représentée comme un flot composé de deux étapes consécutives (figure 3.2) : la génération du code applicatif (GECKO) et la sélection des composants logiciels (APES). La génération du code applicatif part d'un modèle de l'application, d'une description de son déploiement sur le architecture cible et d'un modèle de l'architecture cible, et génère les composants applicatifs correspondants. La sélection des composants logiciels utilise ces composants pour calculer leurs dépendances et ainsi générer leurs graphes de dépendances, qui seront à leurs tours traités par les compilateurs croisés adéquats.

La sortie de ce flot est un ensemble d'objets binaires logiciels, un pour chaque groupe de processeurs homogènes disponible sur la plateforme, chaque binaire contenant la réalisation d'un graphe de dépendance de composants correspondant à un morceau de l'application. Ces composants sont classés en trois catégories : utilisateur, système d'exploitation et abstraction du matériel. Cette classification, présentée au chapitre 4, favorise les critères de portabilité et de mise à l'échelle. Les sections suivantes détaillent à la fois la génération du code applicatif et la sélection de composants logiciels.

3.1 Génération du code applicatif

La génération du code applicatif utilises les éléments passés en entrées — un modèle de l'application, une description de son déploiement et un modèle de l'architecture cible — pour générer un nombre adéquat de composants logiciels correspondant à l'application, ainsi que leurs réalisations respectives. Ces entrées doivent respecter des sémantiques particulières étant donné que le générateur de code, GECKO, a pour mission première de les transformer dans sa propre représentation interne afin d'effectuer les étapes qui conduiront à une génération de code valide. La représentation interne de GECKO, les contraintes sémantique qu'elle impose ainsi que sont *modus operandi* sont détaillés dans les sous-sections suivantes.

3.1.1 Représentation intermédiaire

Afin de pouvoir générer correctement des codes sources pour plusieurs ensembles de processeurs, une hiérarchie structurelle spécifique organisant processeurs, processus, fonctions et communications est requise. Nous avons conçu un modèle objet basé sur la notion de graphe de tâches hiérarchique (*hierarchical task graph* ou HTG) [30], qui réalise cette hiérarchie. Elle définie trois vues : la vue plateforme, la vue réseau de processus et la vue fonctionnelle. La construction de ces différentes vues est autorisée grâce à l'utilisation de contraintes sémantiques fortes imposées sur les trois fichiers d'entrées. Ces contraintes sont discutées dans la section suivante.

La vue plateforme

La vue plateforme représente l'architecture matérielle comme elle est représentée dans le modèle d'architecture matérielle. Seuls les éléments dont le type importe pour la génération de code, à savoir les processeurs, sont représentés : en fonction du type de processeur, d'importantes informations concernant le type du compilateur à utiliser ou bien le type d'optimisations disponibles peuvent être déduites. De plus, les représentations processeurs agissent en tant que conteneurs pouvant réunir les processus auxquels ils ont été dédiés.

La vue réseau de processus

La vue réseau de processus représente le réseau de processus de l'application comme elle est décrite dans le modèle de l'application. Chaque processus est un fil d'exécution indépendant. **Les communications entre les processus sont explicites** : elles doivent utiliser des canaux de communication tandis que l'utilisation de mémoires partagées implicites (souvent par le biais de variables globales) est prohibée. GECKO supporte deux types de processus : les processus hiérarchiques qui encapsulent des graphes de fonctions et les processus virtuels pointant simplement sur des fichiers sources.

La vue fonctionnelle

La vue fonctionnelle représente le graphe fonctionnel de l'application comme il est représenté dans le modèle de l'application. La vue fonctionnel de GECKO peut fidèlement décrire n'importe quelle occurrence de graphe de contrôle et de flot de données (*control and data flow graph* ou CDFG) [44] à condition que le graphe contiennent des fonctions en temps discret. **Les communications entre les fonctions sont implicites** et utilisent généralement de la mémoire partagée locale. Le générateur de code définit trois classes de blocs fonctionnels : les sources, les fonctions et les puits. Ces classes sont utilisées par l'ordonnanceur fonctionnel du générateur afin de produire un code fonctionnellement correct vis-à-vis des dépendances de données du graphe de départ. Néanmoins, le respect de quelconques propriétés d'exécution, telle que la propriété de synchronicité pour les CDFG synchrones, **n'est pas garanti**.

3.1.2 Contraintes sémantiques

Comme nous l'avons présenté dans la sous-section précédente, chacune des trois vues du modèle intermédiaire de GECKO dépend de sémantiques précises que les fichiers d'entrées doivent respecter. La vue plateforme repose sur la notion abstraite de processeur non seulement comme élément de configuration fournissant des informations de compilation, mais aussi comme conteneur regroupant des processus. En conséquence, pour qu'ils soient valides, le modèle d'architecture matérielle doit définir des entités de type processeur et la description du déploiement de l'application doit explicitement associer les processus défini dans le modèle de l'application à ces processeurs.

La vue réseau de processus repose sur la notion de processus indépendants et sur des communications par passage de messages. Ces deux éléments sont la fondation des *réseaux de processus* comme définis dans une *algèbre de processus* (introduite dans [39] et plus généralement présentée dans [5]). Plus simplement, GECKO est capable de traiter n'importe quel

type de réseaux de processus du moment qu'il peut être décrit en utilisant une algèbre de processus, ce qui est le cas de la plupart des configurations de réseaux de processus existants. Par exemple, son modèle de représentation peut décrire un réseau de processus de Kahn (*Kahn Process Network* ou KPN) borné en utilisant des canaux de communication en files d'attente de type *premier entré, premier sorti* (*first in, first out* ou FIFO) bornées, ou bien des processus séquentiels communicants (*communicating sequential processes* ou CSP) en utilisant des canaux de communication à base de rendez-vous. En conséquence, pour qu'il soit valide, le modèle d'application doit contenir un réseau de processus supposé correct pouvant être décrit en utilisant une algèbre de processus.

La vue fonctionnelle repose sur des CDFGs contenant des fonctions en temps discret. Contrairement aux fonctions en temps continu, une fonction en temps discret ne produit pas continuellement un résultat ; son calcul est déclenché lors d'évènements périodiques sur une échelle de temps donnée. Ces évènements périodiques forment la période d'échantillonnage d'une fonction en temps discret et sa sortie est appelée un échantillon. De plus, une fonction en temps discret peut avoir deux comportements. Elle peut soit être purement combinatoire et ses sorties sont disponibles au même cycle d'exécution que ses entrées, soit induire un délai : si ses entrées sont disponibles au temps t, la fonction produit ses sorties au temps $t + n$, avec n la longueur de son délai.

L'utilisation de CDFG en temps discret est un choix sensé : étant donné que les applications logicielles sont exécutées par des machines discrètes, n'importe quel algorithme peut être implémenté en utilisant ce modèle de calcul. En conséquence, si le modèle de l'application définit des comportements pour ses processus, alors ils doivent être décrits comme des CDFG en temps discret. Ces graphes doivent être dépourvus de tous rebouclages (*feed-backs*). De plus, soit le modèle de l'application, soit la description du déploiement de cette application doit contenir des informations concernant les associations entre les processus et les groupes de fonctions.

3.1.3 Génération du composant logiciel

Cette opération comprend trois étapes de génération : le code source de l'application, le descripteur d'interface, et le fichier de commande d'édition de liens.

Code source

GECKO produit le code de chaque processus pour chaque processeur déclaré dans le modèle intermédiaire. Si le processus ne contient pas de graphe fonctionnel, le fichier source est tout simplement copié. Tout d'abord, il produit les commandes d'en-têtes nécessaires ainsi que le prologue du corps du processus. Ensuite, il parcourt la liste des blocs fonctionnels ordonnancés. Les définitions de variables sont générées en premier, suivies par le code des blocs sources, des blocs fonctions et des blocs puits. Enfin, l'épilogue du corps du processus est ajouté. De la même manière, GECKO produit un fichier d'initialisation pour chacun des ensembles de processeurs définis dans le modèle intermédiaire. Tout d'abord, il produit les commandes d'en-têtes nécessaires. Puis, il parcourt les blocs de communications externes et génère la déclaration des canaux. Ensuite, il en va de même pour le prologue de la fonction de démarrage ainsi que pour les commandes d'instanciation des différents processus. Enfin, l'épilogue de la fonction de démarrage est produit.

Descripteur d'interface

Afin de rendre le code source généré reconnaissable comme composant logiciel par l'environnement APES, GECKO a besoin de générer son descripteur d'interface qui contient un résumé des services fournis et requis dans le code source.

Fichier de commande d'édition de liens

Un fichier de commande d'édition de liens est un fichier de configuration parcouru au moment de l'édition de liens par l'éditeur de liens spécifique au processeur cible. A coté des sections généralistes et de la configuration mémoire, plusieurs paramètres de configuration spécifiques au composant peuvent être déclarés dans ce fichier. En conséquence, GECKO a besoin de générer un fichier de commande par composant de l'application. Un tel fichier est généralement dépendant de la chaîne de compilation utilisée. Par exemple, la chaîne de compilation GNU utilise un *ldscript* [31].

3.2 Sélection de composants logiciels

Même si la CBD favorise la réutilisation globale de composants logiciels, elle est encore à trop gros grains pour être complètement satisfaisante. Par exemple, considérons un composant d'ordonnancement comprenant les méthodes suivantes : `elect`, `switch` et `suspend`. Supposons que nous avons à notre disposition l'implémentation d'un ordonnanceur FIFO et que nous voulions implémenter un ordonnanceur à tourniquet. Théoriquement, ne devrions seulement avoir à nous rapporter à l'ordonnanceur FIFO et réécrire la méthode *elect*. En réalité, il nous faudrait dupliquer le code complet de l'ordonnanceur FIFO pour implémenter notre tourniquer. Avec ce type de granularité, la CBD n'est pas utilisable en tant que telle. En effet, plus le code d'un composant grossit et moins devient-il maintenable. Un autre paradigme, appelé programmation orientée objets (*object-oriented programming* ou OOP), offre une bien meilleure granularité. Comparée à la CBD, la OOP garantit une plus grande flexibilité et une réutilisation du code à un grain beaucoup plus fin. Néanmoins, la OOP repose fortement sur des environnements d'exécutions complexes, gourmands en ressources et souvent peu adaptés à du matériel à ressources limitées.

3.2.1 État de l'art

L'encapsulation de morceaux de logiciel afin d'améliorer la flexibilité, la portabilité et la mise à l'échelle d'un projet logiciel n'est pas une nouvelle entreprise. Ceci est particulièrement vrai dans le domaine des systèmes d'exploitation, ou plus précisément les noyaux de systèmes d'exploitation. Ce qui suit récapitule les efforts les plus notables dans ce domaine.

Certains systèmes comme Amoeba [87], Mach [1] ou SPIN [9] essayaient déjà d'encapsuler des fonctionnalités en déplaçant certaines de leurs fonctions dans des processus utilisateurs appelés serveurs, marquant ainsi une fracture avec les architectures monolithiques telle qu'UNIX [77]. Appelés μ-noyaux de première génération, ils étaient encore trop monolithiques et montraient de piètres performances par rapport à leurs ancêtres. D'autres systèmes tels que Windows NT [83] ou eCos [58] définissent un ensemble de fonctions dépendantes du matériel, appelé couche d'abstraction du matériel (*hardware abstraction layer* ou

HAL), afin d'être utilisées partout dans le noyau. Ainsi, les fonctions noyaux ne sont écrites qu'une seule fois puisque le portage d'un tel système ne demande que la réécriture de la HAL pour le processeur et la plateforme cibles. Le temps requis pour cette opération dépend fortement du nombre de fonctions contenues dans cette dernière.

La seconde génération de μ-kernels tels que Exokernel [23] ou bien L4 [52] propose des approches radicalement nouvelles où toutes les fonctions orientées protocoles sont implémentées en tant que serveurs. Seuls certains services obligatoires sont gardés dans le noyau (espace d'adressage, communications inter-processus et ordonnancement des tâches) offrant à cette génération de noyaux des comportements bien meilleurs que ceux de la génération précédente. Par construction, ces μ-noyaux sont relativement modulaires. Des serveurs peuvent être ajoutés ou bien retirés sans modifier le noyau. Ils peuvent être démarrés et arrêtés au besoin et seuls les serveurs requis par l'application sont nécessaires.

Tandis que ces μ-noyaux ont aussi été utilisés pour construire des solutions purement orientées objets [8, 16, 62, 63, 75], ils souffrent de deux inconvénients majeurs : 1) chaque élément du noyau doit être spécialement écrit pour le processeur cible, induisant des temps de portage non-négligeables ; 2) les serveurs sont des processus complets utilisant le mécanisme d'espace d'adressage du noyau, lequel requiert une unité de gestion de la mémoire (*memory management unit* ou MMU)i matérielle. Malheureusement, ce type de matériel est rarement présent sur des MP-SoCs hétérogènes, rendant le concept de μ-noyau pur peu opportun pour ce genre d'architecture.

Des solutions purement CBD comme OSKit [25], Peeble [26] et Think [24] spécifient les services de systèmes d'exploitation de base et les organisent sous forme de composants. Ils utilisent généralement un IDL pour sélectionner les composants requis par l'application et résoudre ses dépendances. Grâce à leur concept strict, ils profitent d'une flexibilité accrue, mais aussi souffrent-ils de la vision à gros grains de leurs composants. En conséquence, ces solutions sont souvent difficiles à porter et à maintenir.

Enfin, l'approche orientée aspects (*aspect-oriented programming* ou AOP) autorise la composition à grains fins de fonctions systèmes lorsqu'elle est couplée avec une claire séparation des préoccupations [55, 56, 81]. Néanmoins, elle repose fortement sur des extensions d'aspects (telles que AspectC++ [57] ou AspectJ [45]), lesquelles n'ont pas été unifiées pour tout les langages de programmation et n'existent que pour certains d'entre eux (comme Java, C++ et C sous forme embryonnaire).

3.2.2 Modèle Composant/Objet

Une solution idéale serait de combiner l'approche légère du CBD avec la réutilisation de code à grain fin offert par la OOP. Ainsi, il serait possible d'apporter une réutilisation du code à grain fin à CBD sous la forme de mécanismes d'héritage et de polymorphisme. Nous avons senti la nécessité de redéfinir un paradigme pour les raisons suivantes : il n'y a aucune autorité notable en matière d'IDL ; les mécanismes d'héritage et de polymorphisme ne sont supporté par aucun IDL ; nous ne voulions pas contraindre le programmeur à utiliser un langage de programmation particulier ; la OOP et la CBD utilisent des mécanisme exécutifs généralement peu compatibles avec des plateformes matérielles limitées en ressources.

Ainsi, nous avons préféré une approche dirigée par les modèles (*model-driven engineering* ou MDE) pour définir nos composants[1]. Le développement dirigé par les modèles est une méthodologie de développement logicielle qui se concentre sur la création de modèles, ou bien

[1] Ce travail a été fait en collaboration avec Guillaume Godet-Bar et Amin Elmrabti.

d'abstractions, se rapprochant des concepts d'un métier particulier plutôt que de concepts algorithmiques ou calculatoires. Son but est de maximiser la compatibilité de systèmes hétérogènes et de simplifier le processus de conception logiciel [80].

Dans le cas de notre paradigme, nommé modèle composant/objet (*object-component model* ou OCM), le MDE permet l'utilisation de l'encapsulation ainsi que de certains mécanismes orientés objets sans utiliser de mécanismes logiciels supplémentaires ni forcer l'utilisation d'un langage de programmation particulier. Pour se faire, chaque composant logiciel est doté d'un descripteur d'interface, ou IFD, et d'un descripteur de réalisation, ou IMD. Le IFD est utilisé pour décrire l'interface du composant et ses dépendances fonctionnelles, et pour générer les fichiers correspondants dans un langage de programmation prédéfini. Le IMD est utilisé pour décrire la réalisation d'un composant. Pour ces descripteurs, nous avons défini leurs règles de construction (comparable à une grammaire), appelées méta-modèle.

3.3 Moteur de construction de graphe

La section précédente a expliqué comment les concepts inclus dans OCM autorisent de manière efficace la réutilisation et la maintenance de morceaux de codes existants. Cette section montre comment les origines CBD d'OCM permettent l'automatisation de la construction du graphe de composants ainsi que la compilation automatique du binaire correspondant.

3.3.1 Théorie

Afin de complètement automatiser ce processus, nous avons développé un algorithme qui repose à la fois sur les dépendances fonctionnelles définies dans le descripteur d'interface d'OCM ainsi que sur l'ensemble complet des éléments disponibles. Cet ensemble, appelé $\mathbb{C}$, est considéré comme un digraphe 4-parti.

Un graphe k-parti est un graphe dont les sommets sont partitionnés en k ensembles disjoints de telle manière que deux sommets d'un même ensemble ne soient pas adjacents. $\mathbb{C}$ est partitionné en quatre ensembles disjoints : l'ensemble des réalisations, l'ensemble des types, l'ensemble des définitions et l'ensemble des méthodes disponibles parmi l'ensemble des composants. Une arête partant d'un élément de l'ensemble des réalisations et pointant vers un élément des autres ensembles représente une relation *require* de l'élément du composant vers l'autre élément. Inversement, une arête partant d'un élément de l'ensemble des types, de l'ensemble des définitions ou bien de l'ensemble des méthodes et pointant vers un élément de l'ensemble des réalisations représente une relation *provide* de l'élément du composant vers l'autre élément.

3.3.2 Construction du graphe et production du binaire

Notre algorithme de construction part de trois éléments : 1) un composant racine appelé $\mathcal{C}$; 2) un ensemble de dépendances appelé $\mathbb{D}$ et initialement vide ; et 3) l'ensemble complet des composants disponibles $\mathbb{C}$. Ensuite, il calcule les dépendance de $\mathcal{C}$ et analyse récursivement chacune d'elles. Une fois que le graphe de composants a été construit, un autre algorithme parcourt itérativement les composants de l'ensemble des dépendance final et compile leurs objets en utilisant le compilateur adéquat. Enfin, l'application finale est produite en appelant l'éditeur de liens adéquat sur l'ensemble des objets préalablement produit.

3.4 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche pour le développement de logiciel embarqué ciblant des architectures MP-SoC. En combinant à la fois un générateur de code et un mécanisme de sélection de composants, notre flot de conception logiciel regroupe de fortes propriétés de conception d'une manière efficace. Nous avons aussi montré que, ensemble, la conception de composants et la programmation orientée objet sont deux approches efficaces pour le développement d'applications embarquées mais que, prises séparément, aucune d'elles n'est suffisante pour satisfaire les contraintes inhérentes aux plateformes embarquées. Nous avons détaillé notre modèle composant/objet, un mélange judicieux entre ces deux mécanismes tirant partie de leur compatibilité théorique pour apporter de hauts niveaux de flexibilité et d'automatisation à la programmation d'applications embarquées.

4
Environnement logiciel

La complexité des plateformes Multi-Processor System-on-Chip (MP-SoC) hétérogènes apporte de nouveaux défis lors de la conception d'un environnement logiciel pour ce type d'architecture. Comme leurs semblables supportant des systèmes embarqués uniprocesseurs ou bien multiprocesseurs homogènes, ces environnements doivent être légers et efficaces. Mais ils ont aussi besoin d'être portables et de pouvoir être aisément mis à l'échelle afin de faciliter le développement d'applications sur ces plateformes.

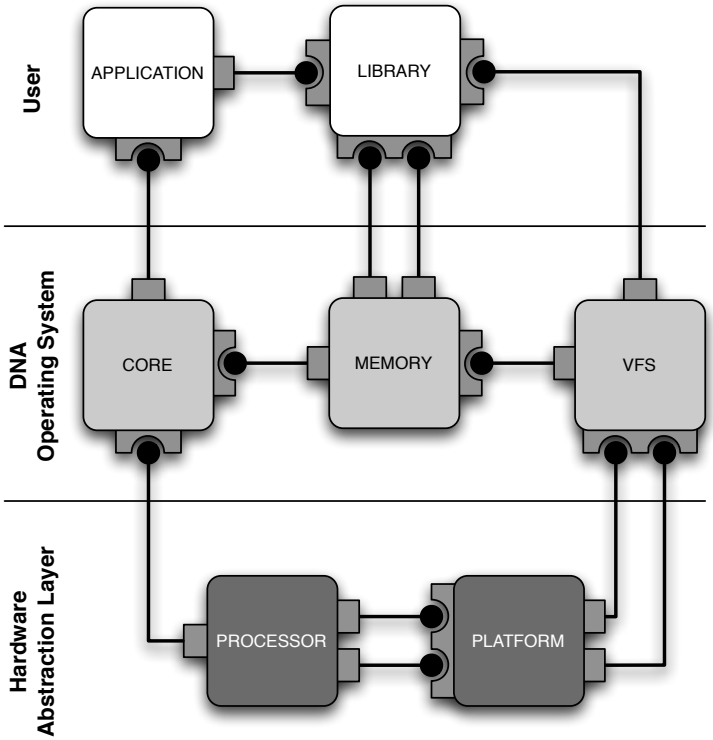


FIGURE 4.1 – Catégories de composants.

Dans le chapitre précédent, nous avons présenté notre flot de conception de logiciel embarqué, utilisant le modèle composant/objet pour fournir de haut niveaux de flexibilité et d'automatisation. Dans ce chapitre, nous présentons l'environnement logiciel que nous avons

développé autour d'Object-Component Model (OCM) afin de proposer de hauts niveaux de portabilité et de mise à l'échelle. Cet environnement et composé de **divers composants logiciels** organisés en trois catégories (figure 4.1) : applications et bibliothèques utilisateurs, le système d'exploitation DNA-OS et la couche d'abstraction du matériel Hardware Abstraction Layer (HAL). Les sections suivantes présentent chacune de ces catégories. Concernant les composants utilisateurs, nous nous concentrerons sur les bibliothèques présentent dans notre environnement.

4.1 Couche d'abstraction du matériel

La HAL fournit le seul et unique accès aux fonctions spécifiques au matériel. Le but de cette couche est d'abstraire les fonctionnalités d'une "plateforme idéale" et d'un "processeur idéal". Pour sa conception, nous nous somme grandement inspiré de la HAL fourni dans eCos [58], bien que nous l'ayons revisité pour supporter des architectures MP-SoC hétérogènes et simplifié afin d'en réduire les surcouts de temps d'implémentation : notre HAL exporte en tout et pour tout vingt-sept fonctions, là où la HAL d'eCos en contient une cinquantaine et le sous-système *arch* de Linux dépasse les cents. Nous avons séparé notre HAL en deux composants OCM : le composant plateforme et le composant processeur.

Du point de vue du matériel, une plateforme consiste en un ensemble de périphériques matériels organisés autours d'infrastructures de communications. Du point de vue du logiciel, une plateforme peut être vue comme une super-entité fournissant des informations qui serait autrement inscrites en dur dans le code du logiciel, telles que le boutisme global et la configuration multiprocesseur globale. Le rôle du composant de plateforme est de fournir ces mécanismes. Il contient en sus des en-têtes de programmes définissant l'organisation fonctionnelle de ses périphériques matériels. Ces en-têtes sont utilisées par les pilotes de périphériques ou bien directement par l'application, au travers des primitives `read`/`write` du processeur.

A partir de simple unités de calculs entiers, les processeurs modernes ont évolués en entités beaucoup plus complexes (contenant une Memory Management Unit (MMU), des caches d'instructions et de données, des unités de calculs flottants, etc.) les rendant compliqués à caractériser d'une manière générique. Ceci est d'autant plus vrai avec les processeurs embarqués, sachant que la plupart d'entre eux peuvent être configurés à volonté. Le rôle du composant processeur est de couvrir la plupart des fonctions partagées par les processeurs modernes, à savoir : le boutisme, les configurations multiprocesseurs, les entrées/sorties, les contextes d'exécution, la synchronisation, les exceptions, la gestion de l'énergie et la gestion de la mémoire et des caches.

4.2 Le système d'exploitation DNA

Le DNA-OS n'est pas qu'un simple système d'exploitation. Lors de la conception de notre organisation logicielle utilisant OCM, nous avons mis en évidence le besoin d'un noyau compatible avec notre HAL, avec un impact léger sur l'emprunte mémoire (en dessous des 32 Ko) et sur les performances globales de l'application, un ensemble de fonctionnalités assez riche pour supporter les bilbiothèques applicatives les plus répandues comme une bibliothèque C complète ou bien un bibliothèque de Portable Operating System Interface [for Unix] (POSIX) Threads (PThreads), et un niveau de compatibilité avancé avec les différents

types d'architectures de processeurs (Reduced Instruction Set Computer (RISC), Digital Signal Processor (DSP), Micro-Controller (uC), ...). De plus, il doit être à base de composants et utiliser OCM.

En considérant tous ces prérequis, il aurait été très difficile d'adapter une solution existante (provenant soit de solutions grand public ou bien de l'état de l'art) à nos besoins. Pour cette raison, nous avons construit le DNA-OS en utilisant une architecture μ-noyau comparable à celle utilisée dans le système d'exploitation BeOS [34] ainsi qu'une forte séparation des préoccupations comme celle mise en avant dans [62,81], à laquelle nous avons injecté le support de la HAL. Ainsi, DNA's Not just Another Operating System (DNA-OS) contient les composants suivants : *Core*, qui multiplexe les mécanismes de base du processeurs ; *Virtual File System*, qui offre une interface compatible POSIX pour la manipulation des fichiers ; et *Memory*, qui offre des services d'allocation mémoire basiques.

4.3 Bibliothèques utilisateurs

Les bibliothèques utilisateurs contiennent des fonctions reposant sur les mécanismes du système d'exploitation et que l'utilisateur peut utiliser directement. Elles peuvent prendre différentes formes : support de langage, fils d'exécutions, support de réseau et ainsi de suite. Notre environnement logiciel propose les bibliothèques suivantes : une implémentation des files d'exécutions POSIX, une implémentation de l'interface DOL, la bibliothèque C Newlib et un port de la pile TCP/IP LwIP.

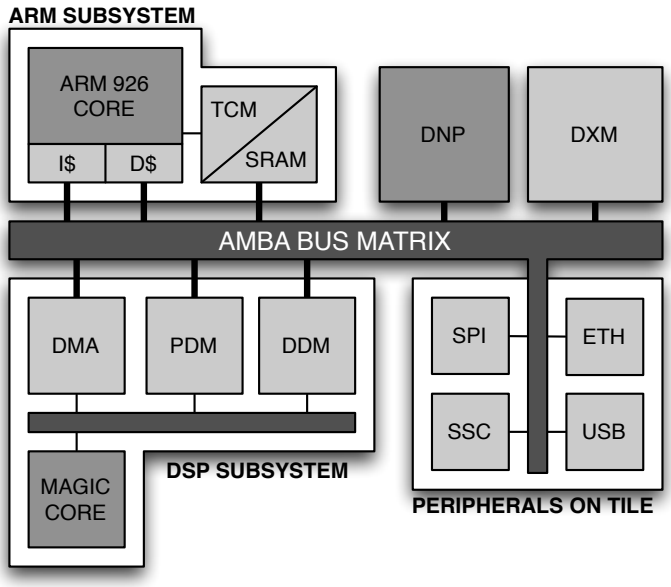


FIGURE 4.2 – La plateforme ATMEL D940

4.4 Utilisabilité et conclusion

Notre environnement logiciel a été utilisé avec succés dans le cadre du projet européen SHAPES. Le but du projet SHAPES [68] était de construire une architecture matérielle composée de plusieurs MP-SoCs hétérogènes et capable d'exécuter des applications complexes

telles que des outils de simulation de physique nucléaire. Chaque MP-SoC contient un processeur ARM 926 ainsi qu'un DSP MagicV (figure 4.2). Il contient aussi plusieurs mémoires et périphériques *on-chip*, ainsi qu'un processeur de réseau garantissant les communications entre les différents SoCs.

Notre contribution, au sein de ce projet, était de fournir le même environnement de programmation sur l'ensemble de la plateforme, peut importe le type du processeur. Le défi de cette opération résidait dans le complexité intrinsèque du SoC. Le processeur ARM peut accéder à une mémoire fortement couplée (*tightly-coupled memory* ou TCM) de 32 Ko en un cycle d'horloge par le biais d'un bus TCM spécifique, tandis que le DSP inclut le cœur de processeur mAgicV, une mémoire de données (DDM, 16 kilo-mots de 40 bits chacun) et une mémoire de programme (PDM, 8 kilo-mots de 128 bits chacun). Le processeur ARM peut accéder directement aux mémoires et aux registres du DSP. Le DSP peut accéder à la mémoire locale du processeur ARM en utilisant son contrôleur DMA intégré.

La même interface de programmation a pu être proposée sur tous les processeurs grâce à l'utilisation de deux réalisations différentes des composants de DNA-OS. Sur le processeur ARM, nous avons utilisé la version complète des composants. Sur le DSP, nous avons utilisé une version limitée des composants, sans mémoire dynamique ni multiprogrammation. Les communications entre les processus situés sur différents processeurs était assurées par des pilotes de communications. Les codes des pilotes utilisés pour chacune des versions du système d'exploitation sont à 90% identiques.

Dans ce chapitre, nous avons présenté un environnement logiciel complet visant à améliorer le développement d'application ciblant des architectures MP-SoC. En utilisant à la fois le paradigme OCM ainsi qu'une stricte séparation entre les éléments dépendants du matériel, le système d'exploitation et les applications, cet environnement apportent de hauts niveaux de portabilité et de mise à l'échelle à la programmation d'applications embarquées. La combinaison de cet environnement avec notre flot de conception logiciel forme une approche de développement solide ayant d'excellents résultats pour chacun des critères fondamentaux.

5
Conclusion

Dans ce manuscrit, nous avons présenté une approche efficace pour le développement rapide de logiciel embarqué ciblant des architectures MP-SoC homogènes et hétérogènes. Basée sur la combinaison d'un flot de conception entièrement automatisé et d'un environnement logiciel à composants, cette approche a été conçue afin de maximiser les critères de flexibilité, portabilité, mise à l'échelle et automatisation, fondamentaux dans le domaine du logiciel embarqué. Nous avons commencé notre dissertation en déclarant que les applications embarquées modernes nécessite une large quantité de puissance de calcul que seules les architectures MP-SoC sont à-même de fournir. Ensuite, nous avons dréssé un comparatifs des solutions industrielles existantes en utilisant les quatres critères fondamentaux comme points de comparaisons. Le première conclusion de ce comparatif fut que, malgrès leur indéniable popularité, aucune des ces solutions n'est capable d'exécuter efficacement du logiciel embarqué sur des architectures MP-SoC.

Dans le chapitre suivant, nous avons présenté notre première contribution : un flot de conception logiciel novateur construit en gardant à l'esprit les principaux avantages des solutions les plus courantes afin d'obtenir de bon résultats dans deux des quatre critères fondamentaux : l'automatisation et la flexibilité. Dans une première section, nous avons démontré comment un premier niveau de flexibilité et d'automatisation peut être atteint grâce à l'utilisation d'un générateur de code. Dans une seconde section, nous avons tout d'abord mis en avant le besoin d'une approche modulaire pour être véritablement flexible et automatique. Ensuite, nous avons présenté OCM, judicieux mélange entre la conception à base de composant et la programmation orientée objet qui, grâce à des mécanismes de sélection automatique de composants, d'héritage de composants et de polymorphisme de réécriture, apporte une réelle contribution à notre flot en matière de flexibilité et d'automatisation.

Au chapitre 4, nous avons présenté notre deuxième contribution : un environnement logiciel à composant utilisant notre paradigme OCM. Tout d'abord, nous avons montré comment une claire séparation entre les applications utilisateurs, le système d'exploitation et la HAL permet d'obtenir de bons résultats en matière de portabilité et de mise à l'échelle. Ensuite nous avons présenté plusieurs composants de notre environnement : les composants HAL, les composants de DNA-OS ainsi que certaines bibliothèques utilisateurs. Enfin, nous avons présenté notre contribution au projet européen SHAPES qui a, grâce à notre environnement logiciel, offert un environnement de programmation homogène aux développeurs d'applications pour une plateforme hétérogène.

Deuxième partie

Unabridged content in English

1
Introduction

Embedded systems have considerably evolved since their first applications in the 1960's when they were merely transistor logic compounds. With the advent and the generalization of the microprocessor in the mid-1980's, most of the hardwired operations were replaced by software applications that would run on a programmable execution unit. In the same time, the very nature of the embedded software evolved.

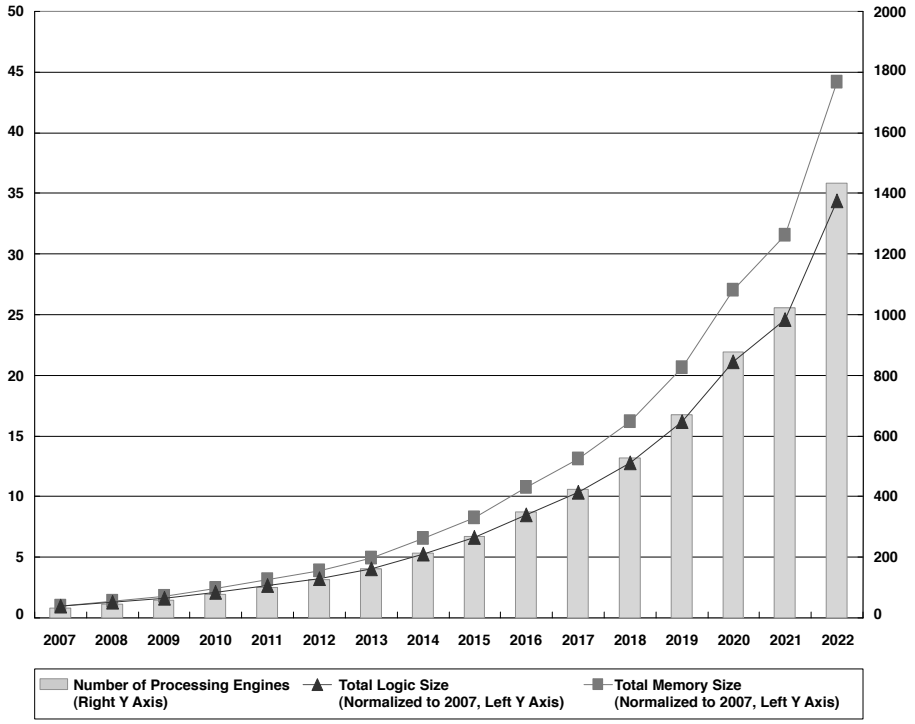


Figure 1.1: International technology roadmap for semiconductors.

The first embedded applications were in many ways comparable to applications designed for early 1980's computers: limited in terms of functionalities and built from scratch, they also had to face strong hardware constraints such as limited amount of memory and low clock frequency. In addition, they also had to rely on assembly languages in order to extract the

very last bit of performance from the processor, whereas modern solutions are much more elaborate.

On the one hand, modern embedded applications contain computation-intensive functionalities, including complex data-crunching algorithms such as multimedia decoder and advanced communications protocols. They are also subject to additional constraints that directly stem from the nature of the targeted platform. The first constraint concerns its electrical consumption. Most of the embedded devices are battery-operated, hence the software developer must include special operations in the application to save energy on the battery. The second constraint is related to the particular dynamic of the embedded system's market. To be competitive, manufacturers must keep the production time and cost extremely low, while optimizing their Time-To-Market (TTM).

On the other hand, modern hardware platforms are more efficient and resourceful. Steadily following the International Technology Roadmap for Semiconductors (ITRS) (figure 1.1), the number of computation cores and the size of the memory integrated on a single chip are constantly increasing (with the psychological barrier of a hundred *on-chip* cores prognosticated for the year 2010). Actually, hardware manufacturers are already investigating Massively Multi-Processor (MMP) solutions that articulate multiple identical processor subsystems around a Network-on-Chip (NoC). But nowadays, embedded appliance designers rely on Multi-Processor System-on-Chip (MP-SoC) architectures to provide the amount of performance required by the applications.

1.1 The system-on-chip architecture

A System-on-Chip (SoC) is a hardware architecture that integrates several electronic components on a single integrated circuit. These components are linked together using an *on-chip* interconnect such as a crossbar, a bus, or a NoC. Among these components are a microprocessor and some memory. The microprocessor is either a Reduced Instruction Set Computer (RISC) processor, a Digital Signal Processor (DSP), or a Micro-Controller (uC). The amount of *on-chip* memory is extremely small ($\simeq$10KB), and the access latency to the data it contains is relatively short (one or two clock cycles).

The other components are usually hardware peripherals such as Input/Output (I/O) controllers, an external memory controller, or hardware accelerators. The first SoCs heavily relied on hardware accelerators to provide software applications with a performance level the sole processor could not possibly offer. As video and telecommunication standards evolve quickly, Intellectual Property (IP) designers no longer consider pure hardware approaches as a viable solution. Consequently, hardware accelerators are progressively replaced by several processors, cheaper to produce and not bound to a particular algorithm.

1.1.1 Homogeneous multiprocessor system-on-chips

A homogeneous Multi-Processor System-on-Chip (MP-SoC) is a SoC that contains several instances of a same, general-purpose, processor core. These cores are accompanied with a small amount of tightly-coupled, *on-chip* memory called local memory. The set {core(s), local memory} is called a core subsystem. The MP-SoC setting is, in many ways, comparable to the Symmetric Multi-Processor (SMP) architecture used in desktop and server appliances. This is its greatest strength, because common approaches can be applied to operate them.

However, it fails to provide a good power/performance ratio to computation-intensive applications, since it is in control-based operations that general-purpose processors score best. The introduction of computation-specific or application-specific cores provide a solution to this problem.

1.1.2 Heterogeneous MP-SoCs

A heterogeneous MP-SoC is a MP-SoC that integrates multiple processor cores of not just one, but many different types (figure 1.2). These cores can be classified into two categories: the General-Purpose Processor (GPP) that can perform any kind of computation fairly well and the Special-Purpose Processor (SPP) that excel in computation-intensive operations. Contrary to homogeneous MP-SoCs, where a single local memory is present for all the cores, each heterogeneous core is accompanied with its own local memory. Each core subsystem can either be directly connected to the main interconnect or have its own communication mechanism linked to the main interconnect using a bridge.

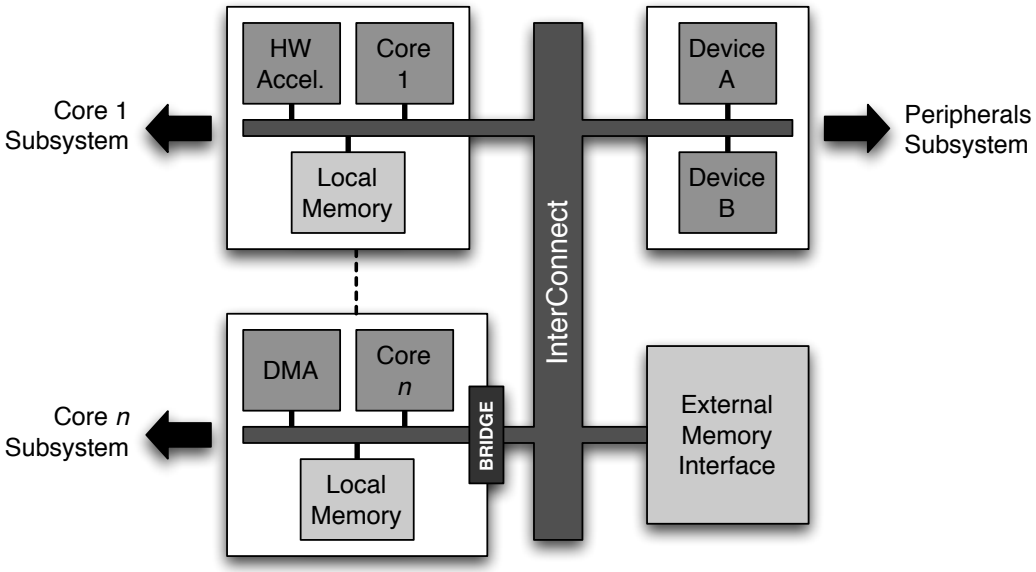


Figure 1.2: Example of Heterogeneous MP-SoC.

The idea that lies behind such a hardware design stems from a *divide and conquer* strategy and the fact that SoCs are usually conceived to execute a specific range of applications. In particular, the MP-SoC design focus on enhancing the execution of software applications. A software application is the implementation, using a software programming language, of a particular algorithm which combines flow control and data processing operations. The same level of performance can be reached with reduced operating frequencies when these operations are spread over specialized cores instead of one or several identical general-purpose cores, resulting in reduced electrical consumption, heat dissipation, and various production costs.

But such a high performance architecture comes with a price: heterogeneous MP-SoC are extremely hard to program efficiently. A programmer developing software for such an architecture will have to cope with different instruction sets, different clock frequencies, different data sizes and types, and non-uniform memory accesses — the software may not have the same view of the hardware's memory map depending on the core on which it is executed. In

addition, each core may have its own mechanism to access locations outside of its subsystem. Not only these differences make the programming of each individual core more difficult, but they also create communication and synchronization problems when two programs running on two heterogeneous cores have to exchange data with one another.

1.2 Purpose of this work

The complexity of the MP-SoC architecture makes traditional programming methods obsolete. No manually-written applications can be developed to efficiently take advantage of such hardware platforms in a reasonable amount of time. Only a method that a) takes into account all the subtleties and capabilities of the architecture at once and b) offers an advanced programming environment that abstracts the complexity from the hardware and automates the code production can achieve acceptable levels of efficiency. The purpose of this work is to provide such a method.

1.3 Organization of this document

The text is structured as follows. Chapter 2 examines the current industrial-class solutions to develop MP-SoC applications in terms of software design flow and presents their related works. Chapter 3 demonstrates how an existing software design flow enhanced with strict component-based and object-based features can improve the development of embedded applications. Chapter 4 gives an extensive description of the APES[1] software framework and the DNA-OS[2] μ-kernel, and shows how a wide range of solutions can be generated. Performance and size measurements of different applications using APES whilst running on both simulated and real MP-SoCs are found in chapter 5. The dissertation closes with chapter 6, which contains conclusions and directions for further works.

[1] Application Elements for SoC.
[2] DNA's Not just Another Operating System.

2
MP-SoC: a programming challenge

Programming a MP-SoC is very different from programming a typical uni-processor machine. This is true at several stages in the design of an application that targets this kind of hardware. At least two characteristics of MP-SoCs have a drastic impact on the design of an application: its multiprocessor nature and its (possible) heterogeneity.

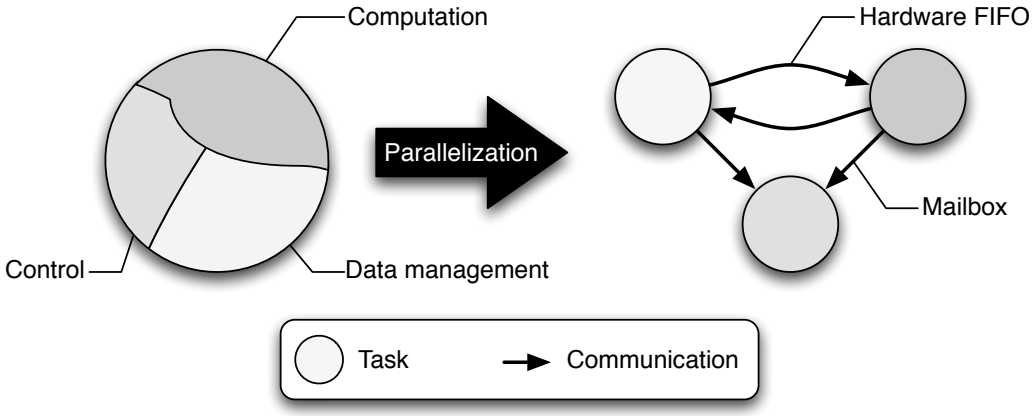


Figure 2.1: Parallelization of an application.

The multiprocessor characteristic implies that the application's algorithm must be parallelized into multiple computation tasks in order to take advantage of the multiple processor cores (figure 2.1). This operation is a delicate one, for badly defined tasks or unwisely selected communications lead to poor overall performance. An application designer can avoid most of the usual pitfalls by answering the following questions:

- **How to balance the computing needs?** Each task must be well defined and its role must be clearly identified.
- **How will the tasks communicate?** The hardware and software communications have to be known and wisely allocated. The software communication primitives have to be wisely chosen and their hardware supports carefully implemented.

The heterogeneous characteristic is probably the most complicated. The difference in the processors' architectures implies that each core executes the software differently from one another. The principal consequence of this statement is that the communications between tasks mapped on two different cores must be mapped onto channels shared by two different control entities. Such channels are not easy to implement, since they require additional synchronization points to perform correctly. An application designer developing for a heterogeneous architecture is confronted with the following issues:

- **Architectural differences:** the cores can have different instruction sets, different word representations (16-bit vs. 32-bit), and/or different endianness.
- **Non-uniform memory access schemes:** the part of the memory accessible from each core may not be the same or may not be accessible the same way (e.g. access latencies, bursts, etc.).

However, these impediments are not usually tackled manually, for most software programmers rely on a source code compiler or at least on a machine code assembler. Therefore, the whole application development process can be depicted as a software design flow (figure 2.2) that takes the following elements as input: a model of the application, a sound knowledge of the hardware platform, and an idea of the hardware/software mapping.

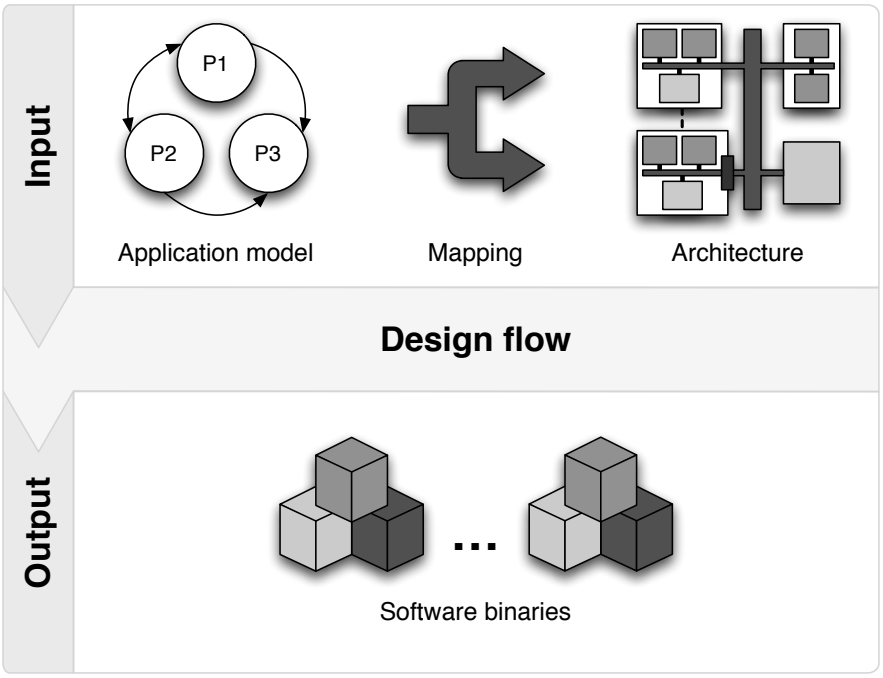


Figure 2.2: Software design flow.

The expected output of this flow is a set of software binaries compatible with the targeted platform's processors that execute the application, structured in three layers: the application layer, the Operating System (OS) layer, and the Hardware Abstraction Layer (HAL) (figure 2.3). This particular layout corresponds to an ideal organization of the key software roles (the application, the system functions, and the hardware dependencies), that offers maximum scalability and portability since the application is not bound to any particular operating system, nor is the operating system dedicated to a specific type or amount of processor/chipset.

The **Application** layer contains an executable version of the application's algorithm. Its implementation depends on the design choices made by the application developer and should not be constrained by the underlying operating system or the hardware, although some constraints (e.g. the amount of available memory) cannot be completely overlooked. It uses external software libraries or language-related functions to access the operating system services, specific communication, and workload distribution interfaces.

The **Operating System** layer provides high-level services to access and multiplex the hardware on which it is running. Such services can be (and are not limited to) multithreading, multiprocessing, I/O and file management, and dynamic memory. It can be as small as a simple scheduler or as big as a full-fledged kernel, depending on the needs of the application. It relies on the HAL to perform hardware-dependent operations, hence ensuring that its implementation is not specific to a particular hardware platform or processor.

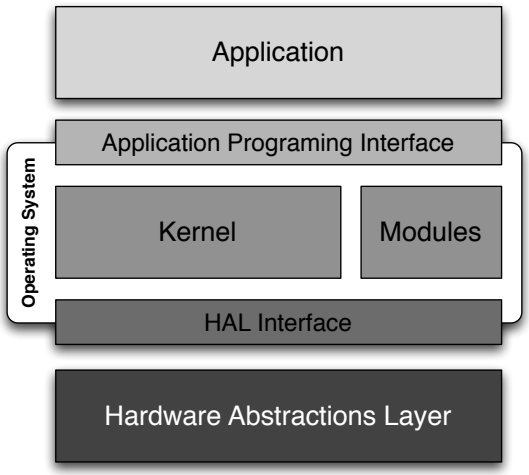


Figure 2.3: Layered organization.

The **Hardware Abstraction Layer** contains several functions that perform the most common hardware-dependent operations required by an operating system. These functions deal with execution contexts, interrupts and exceptions, multiprocessor configurations, low-level I/Os, and so on. This layer usually does not have any external dependencies.

Depending on the complexity and the type of the application, three different kinds of development approaches are considered. The first makes use of a set of hardware-specific, vendor-specific functions provided in what is called a Board Support Package (BSP). The second relies on the services provided by a general-purpose operating system. The third makes use of computation models to automatically generate the application's code. The following sections detail each of these approaches and highlight the design flows used to cope with the parallel and heterogeneous nature of modern SoCs.

2.1 Standalone development

The standalone development approach is the most straightforward solution to start developing a software application on an embedded platform. In its simplest form it only requires a cross-compiler, although software developers usually rely on BSPs. A BSP is a software library that provides access to each hardware device present on the platform. It can be used through

its own Application Programming Interface (API) and is bounded to a specific hardware platform. Contrary to operating systems, BSPs do not provide any kind of system management. They are directly used when something more complex is neither required nor affordable. This is usually the case in the following situations:

- **Small or time-constrained application:** the application is either too simple to require the use of an operating system or its real-time requirements are too high to allow unpredictable behaviors.
- **Limited hardware resources:** the targeted hardware has a very limited memory size or contains only uCs or DSPs, not compatible with generic programing models.
- **Limited human and financial resources:** a more complex software environment cannot be used due to high development costs or a limited workforce.

While the direct use of a BSP may not prevent long-lasting headaches, it can prove to be really useful in cases where the full control over the software — concerning its performances or its final size — is required.

2.1.1 Software design flow

The development of an embedded application using a BSP can be depicted as a flow composed of two uncorrelated processes (figure 2.4): the development of the BSP and the development of the application. Although in a chronological order the development of the BSP comes first, we represented the two steps as parallel events so as to highlight the lack of shared knowledge and competences between the team responsible of the BSP and the one responsible of the application.

Board support package development

The development of a BSP is usually the responsibility of the company that provides the hardware, since a deep knowledge of the architecture is necessary for this operation. It takes the form of several software pieces that provide functions to access and control the hardware devices present on a SoC. It also provides bootstrap codes and memory maps for each of the processors. These libraries are specific to one type of processor and to one type of SoC. Hence, one specific version of a BSP is necessary for each type of processor present on a hardware platform and cannot be used with any other SoC. A BSP is usually proposed in two different types: general-purpose BSPs that export their own specific API and OS-specific BSPs designed to extend the functionalities of an existing, general-purpose operating system.

Application development

The developer of a standalone application needs to be fully aware of the target architecture's details, since an important part of the final source code will be dedicated to the interface between the application and the hardware. This knowledge is so important as to decide the application's mapping and translate it into a source code: the application has to be manually split into several parts dedicated to run separately — albeit cooperatively — on the different processors of the platform.

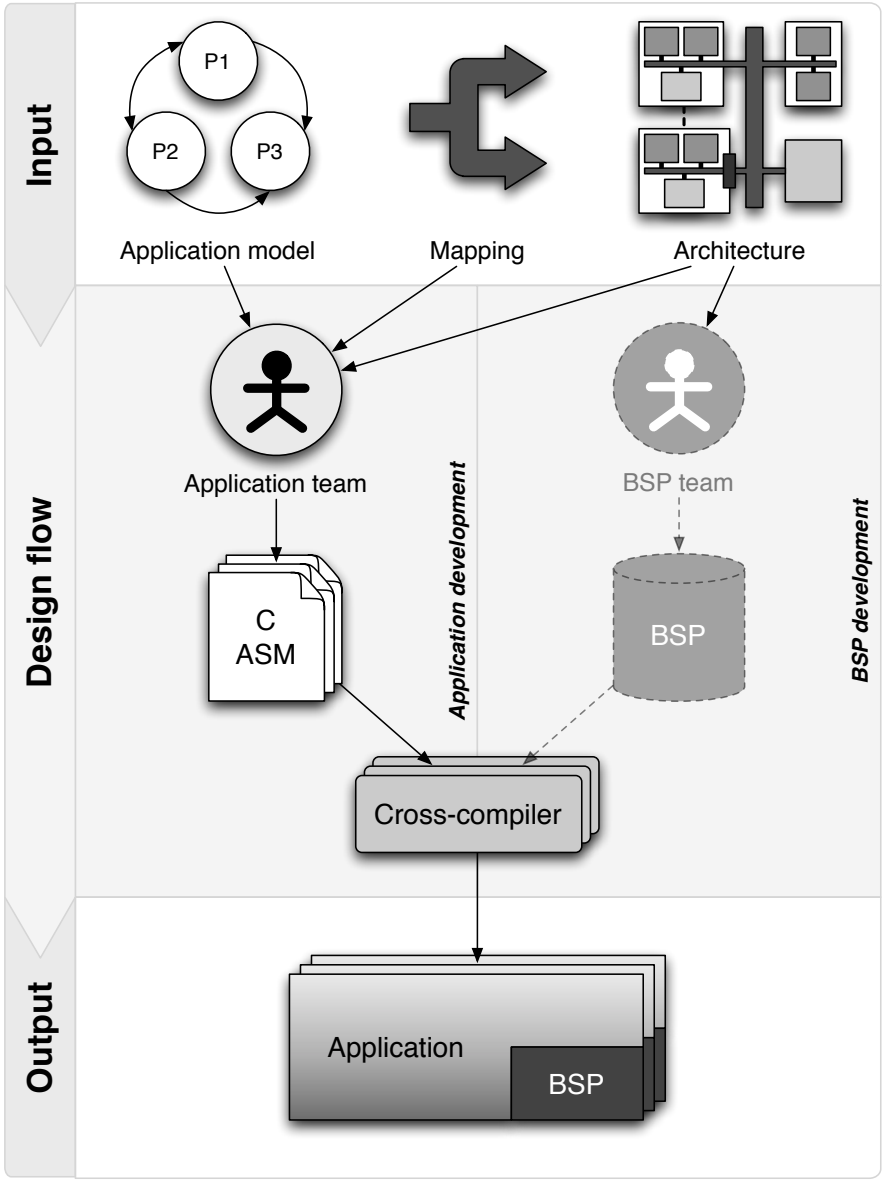


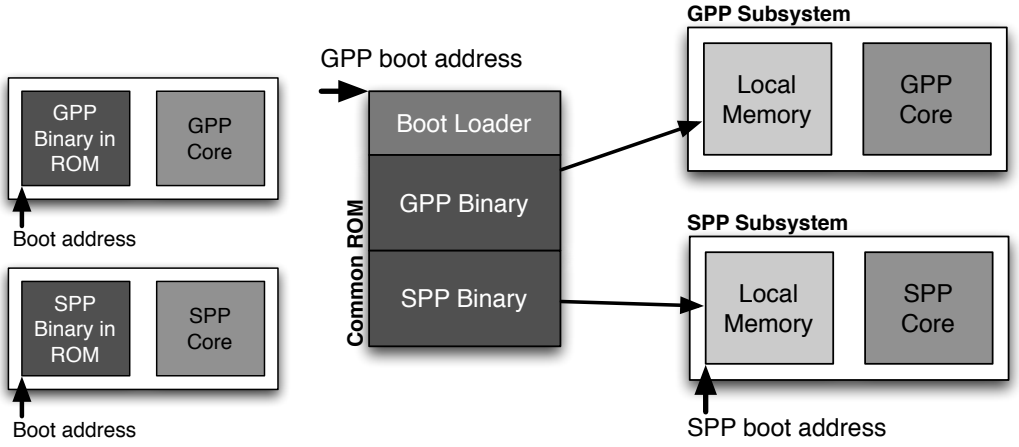
Figure 2.4: Standalone software design flow.

This process is by far one of the most complicated since the full algorithm needs to be thoroughly analyzed in order to get the best partition. It also requires an intensive use of the processor's assembly language and a direct knowledge of memory and peripheral organization. For this reason, the developer may or may not directly interact with a BSP. This decision is usually driven by the overall thoroughness and quality of such a piece of software, two features that vary a lot from one vendor to another. Once the application's source code has been produced, the developer relies on cross-compilers to produce the final binaries. As BSPs, cross-compilers are generally provided by the hardware vendor.

Debug and deployment

In BSP-based configuration, the debugging of the application is done directly on the hardware using boundary-scan emulators connected on the Test Access Port (TAP) of the proces-

sor (if one is available) coupled with an external debugger [41, 90]. The final step deals with the application's booting sequence. Although it closely depends on the hardware architecture, two methods can be distinguished: a) each binary is placed on a Read-Only Memory (ROM) or an Electrically Erasable and Programmable Read-Only Memory (EEPROM) specific to each processor, and b) all the binaries are placed on the same ROM device.



(a) Multiple ROMs (b) Single common ROM

Figure 2.5: BSP-based boot-up strategies.

In method a), as depicted in figure 2.5(a), all the processors are independent from each other and autonomously start at the address on which their local ROM is mapped. In method b), as depicted in figure 2.5(b), one processor is designated to boot first while the others are put in idle mode. This processor is responsible for dispatching the binaries and starting the remaining processors.

2.1.2 Existing works

On the one hand, vendors such as Altera, Xilinx, Tensilica, etc. provide general-purpose BSPs coupled with Integrated Development Environment (IDE) dedicated to the development of applications for their hardware platforms. They also provide software libraries containing standard C functions, network management functions, and basic thread management functions. On the other hand, vendors such as Texas Instruments, Atmel, Renesas, etc. provide OS-specific BSP for systems such as Windows Mobile or Linux. These BSPs are not supposed to be used outside of their specific operating system target, and they are usually prepackaged to be directly installed in the OS's development environment.

2.2 General-purpose operating system

A General-Purpose OS (GPOS) is a full-featured operating system designed to provide a wide range of services to all types of applications. These services usually are (and not limited to) multiprocessing, multithreading, memory protection, and network support. GPOSs are not specifically designed to operate MP-SoCs. They usually are merely adapted from other computing domains such as desktop solutions or uni-processor embedded solutions in order to provide a development environment similar to what software developers are generally used

to. They are used when there are sufficient hardware resources and when a more specific system solution is not required. This is usually the case in the following situations:

- **Portability or limited knowledge of the target hardware:** the application needs to be adapted to multiple hardware targets and gathering a perfect knowledge for each of them is not feasible (of course, the port of the GPOS on the targeted hardware must be available).
- **Application complex but not critical:** the application requires high-level system services such as thread management or file access and does not have particular performance constraints.
- **Limited time resources:** for different reasons, the development time of the application is limited. Hence additional developments, required by the main application, must be kept to a minimum. In addition, common and well-known programming environments are preferred in order to save the time necessary to learn new software programming methods.

One of the principal advantages of using a general-purpose operating system is the availability of a large amount of resources from its community (such as external support, existing hardware drivers, etc.) that can greatly speed-up the software development process. The other advantage is the availability of a well-established development environment, containing many libraries for application support and several tools such as compilers, profilers, and advanced debuggers.

However, in the GPOS-based approach heterogeneous hardware platforms are assimilated to standard hardware configurations, where the GPP is seen as the master processor and the SPP is seen as a co-processor dedicated to specific tasks such as video or audio decoding. Strong hypothesis are made concerning the GPP capabilities for the GPOS to run correctly: 1) it is supposed to be the only processor to have a complete control over the hardware, 2) it has no limitation in terms of addressing space, and 3) it can decide whether or not a SPP can be started. As a consequence, only a restrained subset of MP-SoC contains platforms complex enough to run a GPOS.

2.2.1 Software design flow

In a way similar to the standalone approach, the development of an embedded application using a GPOS can also be depicted as a flow composed of two uncorrelated processes (figure 2.6): the development of the GPOS and the development of the application.

General-purpose operating system development

A GPOS is a standalone software binary, running in supervisor mode, that provides services to applications, running in user mode, through system calls. It usually requires hardware support for atomic operations and virtual memory management, and consequently is dedicated to run on a GPP. Another consequence of these requirements is that the other processors of the hardware platform are not seen as such. They are seen as hardware devices that can only be accessed (hence programmed) through device drivers of the GPOS. This particularity radically changes the programming model of the application as compared to the programming model of the previous approach.

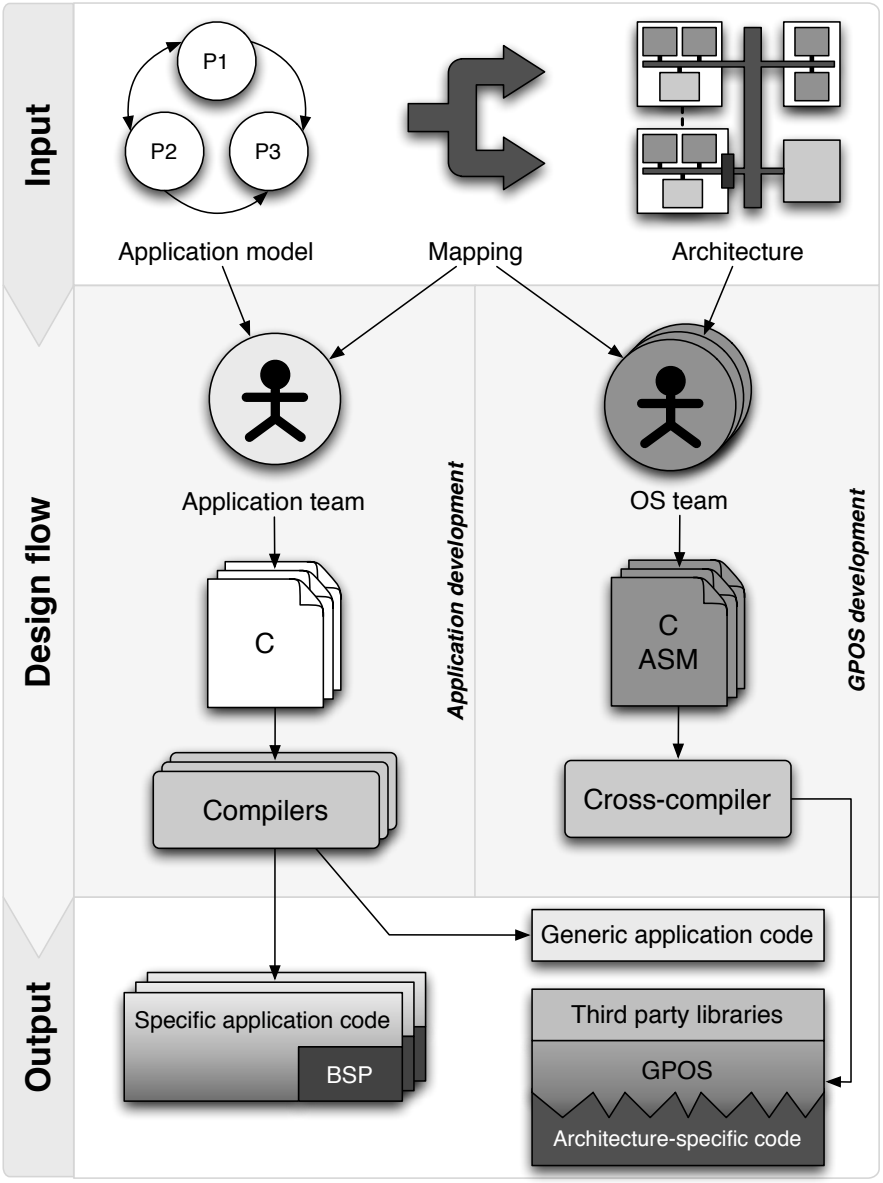


Figure 2.6: GPOS design flow.

A GPOS has its own development team, which can exceed hundreds of people. This team is usually divided in work groups, each dedicated to a particular area of the operating system. Communication between these groups is usually kept to a minimum, while communication with external application developers is simply non-existent[1].

Creation of the application

Although not as dependent on a deep knowledge of the hardware as the previous approach, the application developers need to go through the whole process of manually splitting the

[1]This assertion is not true for every solution: while a little rough on the edges, some open-source projects offer basic one-to-one support with external application developers, and if the developers are wealthy and willing to spend gigantic amounts of money, some commercial projects offer dedicated support solutions.

application into parts that do not need specific accelerations and parts that can take advantage of the SPPs. The parts of the application dedicated to run on the GPP are developed using tool-chains specific to the processor and to the GPOS. They cannot access directly the peripherals and, although the use of assembly code is allowed, only *mnemonics* available in user mode can be used. *Au contraire*, they are bound to either use GPOS-specific, third-party libraries or directly use the kernel's system calls. The parts of the application dedicated to run on the SPPs are mainly developed using a BSP-based method as explained in the previous section.

Communications between different parts of the application are performed by specific device drivers compatible with the host GPOS. Unlike the previous approach, where these communications can be very efficient, the operating system is required to communicate with the SPPs which has a critical impact on the application performance. As a consequence, a *master-slave* solution — where the GPOS has a full control over the SPPs — is usually favored. In this case, communications between the different parts of the application are reduced to a bare minimum: 1) gathered in different firmware, sets of functions are dumped into the *SPPs* through the device drivers; 2) the GPOS control the call to these functions; 3) the code on the SPP deals with fetching the inputs and dispatching the result in the correct destination buffer.

Debug and deployment

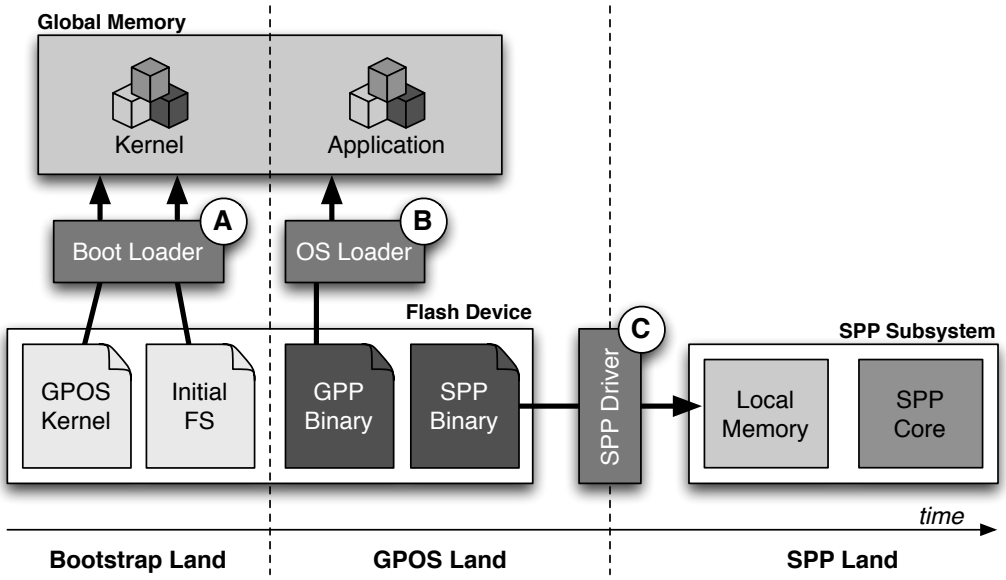


Figure 2.7: GPOS-based boot-up sequence.

The debugging of an application that partially relies on a GPOS is slightly more difficult than when only BSPs are used. Although the method is the same for the parts of the application running on the SPPs as when a BSP is directly used, two methods for the parts running on the GPP are available: one can use either an external debugger connected to the TAP port of the platform or an internal debugger running on the GPOS. Unfortunately, none of these approaches is truly efficient. In the first approach, the external debugger must be able to load and boot the kernel of the GPOS and, in that case, not only the application is being debugged but the whole operating system as well, increasing the complexity of the operation by a hundredfold. In the second approach, only the application is being debugged. However,

if something corrupts the kernel of the GPOS (such as a bug in a driver or a bad operation from one of the SPP) then the whole operating system will crash, including the debugger and the application that is being debugged.

The boot-up sequence of the GPOS-based approach heavily relies on the hypothesis that the GPP is the master of the board and the that SPPs have not started until the general-purpose operating system initiates them. The binaries, including a boot loader, the GPOS, and its (generally huge) initial file system, can be placed either on an internal ROM or on an internal Flash Memory device. This choice is closely related to the memory space required by the GPOS and its initial file system. When the hardware is powered-up, the GPP executes the boot loader which is in charge of loading the GPOS kernel into the global memory and booting the operating system (figure 2.7(A)). Then, once the GPOS is booted it loads the GPP-specific part of the application into the global memory (figure 2.7(B)) and the SPP-specific parts of the application onto the SPPs local memories using the SPP device drivers. Finally, the GPOS starts the SPPs as well as the GPP-specific part of the application (figure 2.7(C)).

2.2.2 Existing works

VxWorks [91] is a real-time, closed-source operating system developed and commercialized by Wind River Systems. It has been specifically designed to run on embedded systems. It runs on most of the processors that can be found on embedded hardware platforms (MIPS, PowerPC, x86, ARM, etc.) and its micro-kernel supports most of the modern operating system services (multitasking, memory protection, SMP support, etc.). Applications targeting this operating system can be developed using the Workbench IDE. VxWorks has been used in projects such as the Honda Robot ASIMO, the Apache Longbow helicopter, and the Xerox Phaser printer.

Windows CE [65] is Microsoft's closed-source, real-time operating system for embedded systems. It is supported on the MIPS, ARM, x86, and Hitachi SuperH processor families. Its hybrid kernel implements most of the modern system services. Applications targeting this operating system can be developed using Microsoft Visual Studio or Embedded Visual C++. Windows CE has been used on devices such as the Sega Dreamcast or the Micros Fidelio Point of Sales terminals.

QNX [46] is a micro-kernel based, closed-source, UNIX-like operating system designed for embedded systems developed and commercialized by QNX Software System. It is supported on the x86, MIPS, PowerPC, SH-4, and ARM processor families. Its kernel implements all the modern operating system services and supports all current POSIX API. It is renown for its stability, its performance, and its modularity. Applications targeting this operating system can be developed using the Momentics IDE, based on the Eclipse framework.

eCos [58] is a real-time, open-source, royalty-free operating system specifically designed for embedded systems initially developed by Cygnus Solutions. It mainly targets applications that require only one process with multiple threads. It is supported on a large variety of processor families, including (but not limited to) MIPS, PowerPC, Nios, ARM, Motorola 68000, SPARC, and x86. It includes a compatibility layer for the Portable Operating System Interface [for Unix] (POSIX) API. Applications targeting this operating system can be developed using specific cross-compilation tool-chains.

Symbian-OS [37] is a general-purpose operating system developed by Symbian Ltd. and designed exclusively for mobile devices. In 2008 a new, independent, non-profit organisation

called the Symbian Foundation was established with purpose of publishing and maintaining the Symbian-OS source code. Based on a micro-kernel architecture it runs exclusively on ARM processors, although unofficial ports on the x86 architecture are known to exist. Applications targeting this operating system can be developed using an Software Development Kit (SDK) based either on Eclipse or CodeWarrior.

μC-OS/II [47] is a real-time, multitasking kernel-based operating system developed by Micrium. Its primary target are embedded systems. It supports many processors (such as ARM-7TDMI, ARM926EJ-S, Atmel AT91SAM family, IBM PowerPC 430, ...) and is suitable for use in safety critical systems such as transportation or nuclear installations.

Mutek [70] is an academic OS kernel based on a lightweight implementation of the POSIX Threads (PThreads) API. It supports several processor architectures such as MIPS, ARM, and PowerPC, and application written using the PThreads API can be directly cross-compiled for one of these architectures using Mutek's API.

Linux [12] is an open-source, royalty-free, monolithic kernel first developed by Linus Torvald and now developed and maintained by a consortium of developers worldwide. It was not designed to be run on an embedded device at first but due to its freedom of use, its compatibility with the POSIX interface, and its large set of services it has become a widely adopted solution (as well as in the mobile market, with solutions such as Android [14]). It supports a very large range of processors and hardware architectures, and it benefits from an active community of developers. Soft real-time (PREEMPT-RT), hard real-time (Xenomai [29]), and security (SE-linux [59]) extensions can be added to the mainline kernel. Applications targeting operating systems based on this kernel can be developed using specific cross-compilation tool-chains.

2.3 Model-based application design

Model-based application design stems from the System-Level Design (SLD) approach, where the implementation of an application is decoupled from its specification. Formal mathematical models of computation are used instead of standard programming language to describe the application's behavior. Software dependences such as specific libraries of Real-Time (RT)OS functionalities can also be modeled. This allows the software designer to perform fast functional simulations and validate the application early in the development process. This solution is generally used when the application's behavior needs to be thoroughly verified and the validation of the software needs to be fast and accurate. This is usually the case in the following situations:

- **Safety-critical applications:** the application will be embedded in high-risk environments such as cars, planes, or nuclear power plants.
- **Time-critical applications:** each part of the application needs to be accurately timed in every possible execution case.

It is also used in industries that must rely on external libraries that have earned international certifications, such as DO-178B/EUROCAE ED-12B [79] for avionics or SIL3/SIL4 IEC 61508 [42] for transportation and nuclear systems.

The development of an application that uses this approach starts with the description of the application's algorithm in a particular model of computation. This model of computation

must fit the computational domain of the algorithm and it must be supported by the code generation tool. The most widely used models of computation are: Synchronous Data Flow Graph (SDFG), Control Data Flow Graph (CDFG), Synchronous and Control Data Flow Graph (SCDFG), Final State Machine (FSM), Kahn Process Network (KPN), and Petri Nets (PN). The execution can be time-triggered, resulting in the repeated execution of the whole algorithm at any given frequency. Or it can be event-triggered, causing the algorithm to be executed as a reaction to external events.

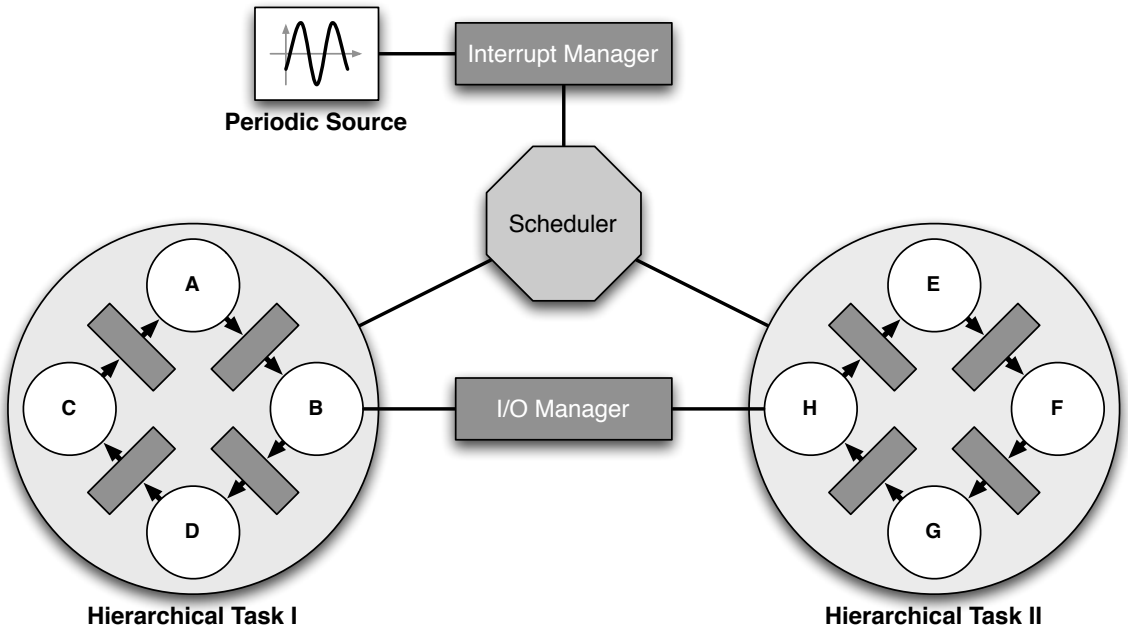


Figure 2.8: Tasks graph with RTOS elements.

Next, the model can be organized in a Hierarchical Task Graph (HTG) [30], where different parts of the application are encapsulated into tasks. RTOS elements can be added to the model and connected to the tasks in order to extract information and timings about the behavior of the whole software organization early in the development process (figure 2.3). These elements can be RT schedulers, interrupt managers, or I/O managers [69]. They can also be adjusted (e.g. the scheduling policy can be changed) to fit the application requirements. Finally, the model may or may not include the mapping of the expressed tasks onto the available processors. In order to produce executable binaries, it is used as an input of the software design flow described below.

2.3.1 Software design flow

The development or, more accurately, the code production of a model-based application can be depicted as a flow composed of two steps (figure 2.9): the development of platform support and 3rd-party libraries, and the application code generation. Contrary to the previous flows, these steps take place in an automated environment (the application generator), bringing more cohesion and homogeneity to the whole process: given that an element of the flow can be either produced by or integrated into the generator, each of them is inherently compatible with the others.

Platform support library development

The development of a Platform Support Library (PSL) is the responsibility of the company that provides the generation tool. A PSL is, in substance, similar to a BSP since it requires a deep knowledge of the target hardware architecture. The main difference is the completeness of the hardware support: where a BSP tends to provide an API for most of the hardware devices present on the platform, a PSL only covers the needs of its related generator. Like BSPs, PSLs are specific to one type of processor and to one type of SoC.

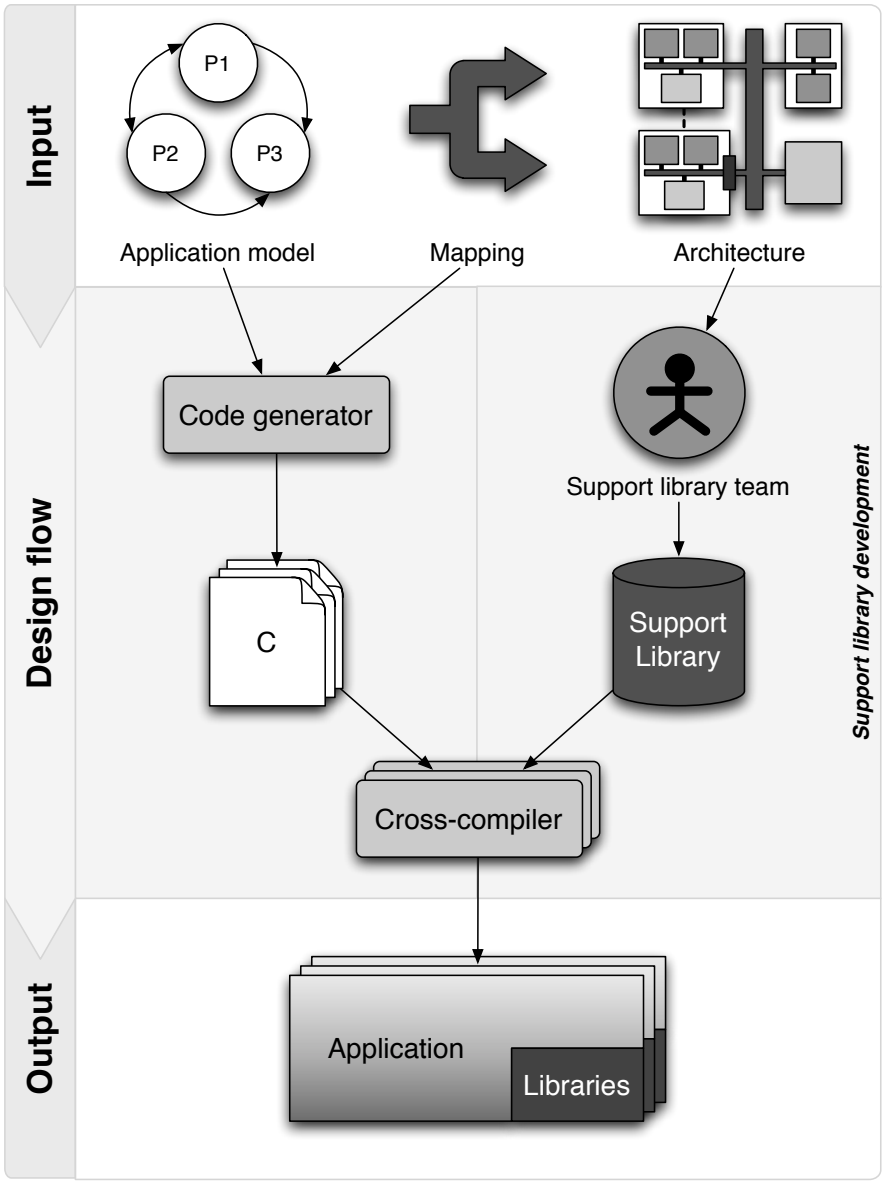


Figure 2.9: Model-based software design flow.

3rd-party libraries provide extra functionalities such as RTOS capabilities or advanced signal processing. They are usually developed by people external to both the application and the PSL developers whom may not have prior knowledge of the targeted platform. However, the simple fact that they are integrated into the generation environment guarantee their compatibility with the application.

Application generation

Starting from an application model designed using a supported model of computation, the code of the application is generated in a language supported by the code generator[2]. Operating system and communication elements are considered as external libraries, which export their programming interfaces to the code generator. The tasks defined in the model are encapsulated in execution threads compatible with the supported operating system. Communications between the tasks are performed using functions from one of the external communication libraries.

Debug and deployment

The model-based application design theory guarantees that the produced binary for a targeted platform will behave the same way as its model does. This is a reason why the debugging facilities of most existing model-based design environments are focused on the model itself, and not the produced code. This assertion proves to be true in most cases targeting relatively simple hardware platforms and using only certified libraries.

However, the process becomes a lot more complex when the targeted platform contains more than one processor or when uncertified libraries are used. As a result, most model-based design environments do not provide either support for multiprocessor architectures (either homogeneous or heterogeneous) or On-Chip Debugging (OCD) capabilities. Therefore, developers interested in either one of these are forced to use methods comparable to the standalone development approach. This statement is also valid for the application deployment.

2.3.2 Existing works

SPADE [53], Sesame [71], Artemis [72], and Srijan [21] start with functional models in the form of Kahn Process Network (KPN). These approaches are able to refine automatically the software from a coarse grained KPN, but they require the designer to determine the granularity of processes, to specify manually the behavior of the tasks, and to express explicitly the communication between tasks using communication primitives.

Ptolemy [13], Metropolis [6], and SpecC [15] are high-level design frameworks for system-level specification, simulation, analysis, and synthesis. Ptolemy is a well-known development environment for high-level system specification and simulation that supports multiple models of computation. Metropolis enables the representation of design constraints in the system model. The meta-model serves as input for all the tools built in Metropolis. The meta-model files are parsed and developed into an Abstract Tree Syntax (AST) by the Metropolis front-end. Tools are written as back-ends that operate on the AST and can either output results or modify the meta-model code.

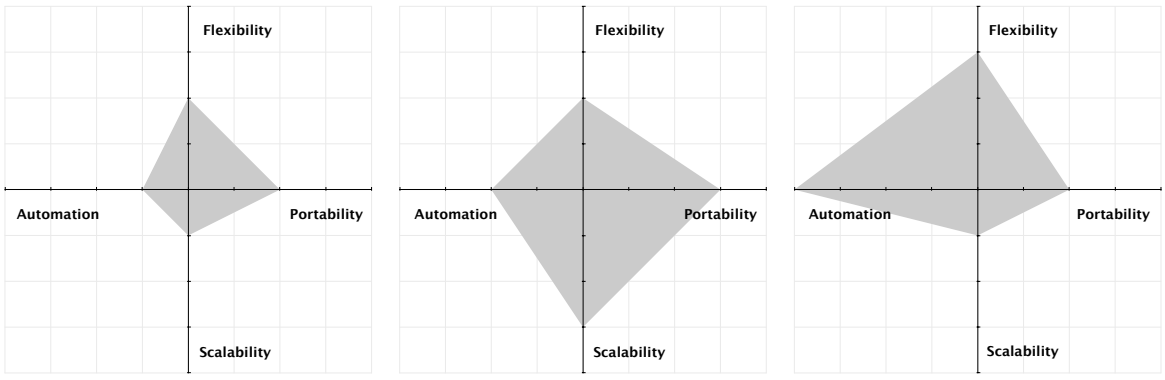
MATCH [7] uses Matlab descriptions, partitions them automatically, and generates software codes for heterogeneous multiprocessor architectures. However, MATCH assumes that the target system consists of Commercial Off-The-Shelf (COTS) processors, DSPs, Field-Programmable Gate Array (FPGA), and relatively fixed communication architectures such as ethernet and VME buses. Thereby MATCH does not support software adaptations for different processors and protocols.

[2] It is important to note that only model of computation supported by the generator can be used in this approach, disqualifying *de facto* applications written using standard programming languages.

Real-Time Workshop (RTW) [43], dSpace [89], and LESCEA [35] use a Simulink model as input to generate software code. RTW generates only single-threaded software code as output. dSpace can generate software codes for multiprocessor systems from a specific Simulink model. However, the generated software codes are targeted to a specific architecture consisting of several COTS processor boards. Its main purpose is high-speed simulation of control-intensive applications. LESCEA can also generate multithreaded software code for multiprocessor systems. The main difference with dSpace is that it is not limited to any particular type of architecture.

2.4 Analysis

In this section, we present an analysis of the pros and cons of the previously presented approaches. This analysis is based on four criteria: flexibility, scalability, portability, and automation. The flexibility criterion shows how an approach favors software component reuse and optimizes the size of the final binaries. The scalability criterion expresses the ability of an approach to be scaled up or down depending on the targeted platform (e.g. increased or reduced number of processors, one or multiple chips, and so on). The portability criterion evaluates the cost to adapt a specific approach to several different target platforms. The automation criterion indicates the ability of an approach to reduce the time and cost parameters of its software development cycles by using automated processes.



(a) Standalone (b) GPOS-based (c) Model-based

Figure 2.10: Software design approach statistics.

2.4.1 Standalone approach

The standalone development approach suffers from its lack of scalability and automation (figure 2.10(a)). Although it features honorable scores in terms of flexibility and portability, this approach is clearly reserved to small projects with long life cycles that require complete control over each element of the hardware.

2.4.2 GPOS-based approach

The GPOS-based development approach features good scores in terms of portability and scalability, thanks to the large numbers of supported hardware architecture and software mechanisms (figure 2.10(b)). However, it demonstrates low scores in flexibility and automation.

In fact, the monolithic architecture of GPOSs tends to produce overweight, sub-optimized binaries and, although they offer typical development environments, their associated development approach is often merely hand-programming. Therefore, projects that require strong operating system support, and which are not too strict in term of hardware resources (such as set-top boxes or *smartphones*), would benefit from this approach.

2.4.3 Model-based approach

Thanks to its automated code generator and its library mechanism, the model-based development approach does extremely well in terms of automation and relatively well in term of flexibility, although its incompatibility with applications not designed using supported models of computation can be problematic. It behaves poorly in terms of scalability and portability. This is not a surprising result, since automated solutions usually rely on well-known architecture templates to generate their code. In addition, since the produced binary and the initial model must behave identically, any indeterminism is avoided. As a consequence, only simple, uni-processor hardware platforms are usually supported. Therefore, safety- or time-critical projects (such as software FSM or event-driven systems that are developed in the automotive or the avionic industries) can take advantage of these environments.

2.4.4 Summary

While each of these approaches brings its share of advantages — except the standalone approach, far too primitive to be of any use – none of them completely fits our needs. The GPOS-based approach offers several high-level services and a well-known environment to the programmer but clearly cannot take advantage of heterogeneous configurations. Concerning the GPOSs themselves, none of them have been designed to operate heterogeneous multiprocessor configurations and they would require too much rework to be able to do so: while this operation is not possible on closed-source operating systems, it would be a herculean task to apply on open-source operating systems. The support for heterogeneity has to be integrated early in the design of an operating system. The model-based approach barely supports multiprocessor platforms, let alone heterogeneity. In addition, most of the code generation solutions are part of closed-source, expensive environments that cannot be modified to the extent required by the targeted hardware. As for the GPOS-based approach, the integration of heterogeneity has to be done early in the design of the generators.

2.5 Contribution

This dissertation shows that complex, embedded software applications can effectively operate heterogeneous MP-SoC with respect to flexibility, scalability, portability, and Time-To-Market. It presents an improved embedded software design flow that combines an application code generator, GECKO, and a novel software framework, APES, to achieve a high level of efficiency. Our contribution is twofold: 1) an improved embedded software design flow with several tools that enable the automatic construction of minimal and optimized binaries for a given application targeting a given MP-SoC as presented in chapter 3, and 2) a modular and portable set of software components that includes traditional operating system mechanisms as well as the support for multiple processors as presented in chapter 4.

3
Embedded software design flow

In the previous chapter, we learned that none of the existing software approaches satisfy all four criteria — flexibility, scalability, portability, and automation — at the same time, while an optimal approach to develop software on MP-SoC should meet them all:

Flexibility: The more complex the application gets, the more software components are likely to be reused. In a same way, software developers are likely to prefer small, optimized binaries that contain only what is necessary rather than bloated ones.

Portability: Like features must not be implemented twice for two different platforms or two different processors if it is not fundamentally hardware-dependent. Identically, porting an application from one hardware platform to another should be seamless and pain-free.

Scalability: From one hardware architecture to another, the number of processors may vary or a device may have been added or modified. The migration to the new architecture should only be a matter of reconfiguration and not reprogramming.

Automation: A single application may require several operating system features, third-party libraries, device drivers, file system modules, etc. The selection and specialization of these different elements should be automated. In addition, a developer may want to use a level of abstraction higher than plain C code. As a consequence, the source code corresponding to its application should be automatically generated and compatible with its targeted platform.

In this chapter, we present a software design flow that, by joining both an application code generator and a software component management mechanism, combines the flexibility and automation criteria. One of the key concepts of our approach is the Component-based Development (CBD) paradigm, that defines the notion of software component. A software component is a software organization that contains the implementation of different software functionalities (figure 3.1). It combines an interface that exports the methods it provides and the methods it requires with a shell that contains the implementations of the methods it provides. The link between a method provided by a component and required by another is called a dependence.

This organization is usually built with a specific context in mind where the methods refer to a specific type or a specific abstraction in order to minimize unfortunate overlaps. Each method can be either publicly available (i.e. appropriate for external use by other components) or private to the component. Public methods can be accessed using their signatures.

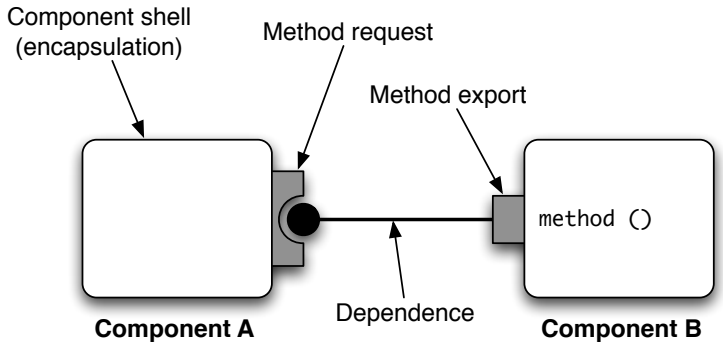


Figure 3.1: Structure of a software component.

The component interface contains the union of the signatures of the component's public methods as well as the union of the signatures used in the component's implementation. The presence of the required signatures is mandatory to enable the resolution of the component dependences. In addition, a component's interface is, from a point of view exterior to the component, the only access to its methods. As a consequence, its implementation can be completely opaque towards other components. This statement is valid since no knowledge concerning the internal mechanisms of a particular implementation is shared, hence forcing an external component to rely on the explicit functionality of the method and not its potential side effects.

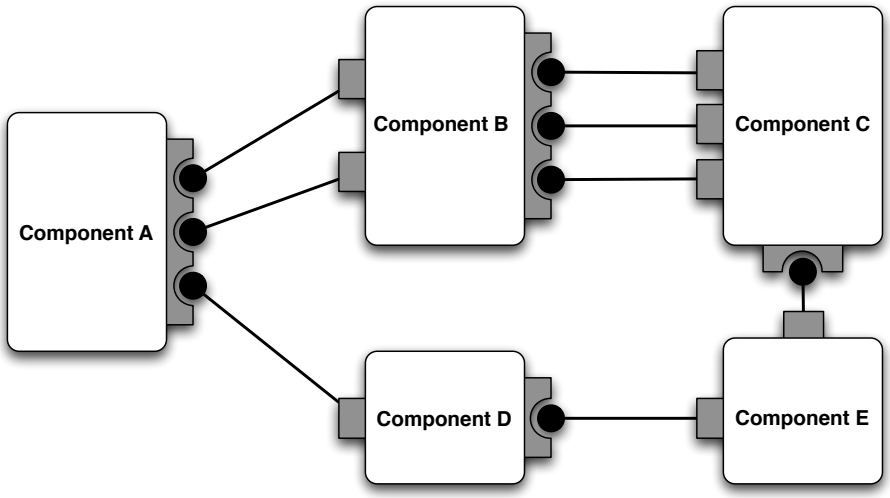


Figure 3.2: Example of a dependency graph.

Once defined, the interfaces need to be declared in order to be exploitable. Historically, developers rely on an Interface Definition Language (IDL) to perform this task. Languages of this kind describe the interface of a component without prior knowledge of the language used to actually implement the interface's behavior [38, 48, 82]. Once the descriptions are written, they are usually compiled in an application that can later be used to generate the

interface specifications compatible with the programming language used in the implementations. These interfaces not only explicitly improve the reuse of the components but they also accelerate the programming process: for a given component, with the help of primitives such as *provide* or *require*, a software dependency graph (figure 3.2) can be automatically generated and its corresponding binary constructed.

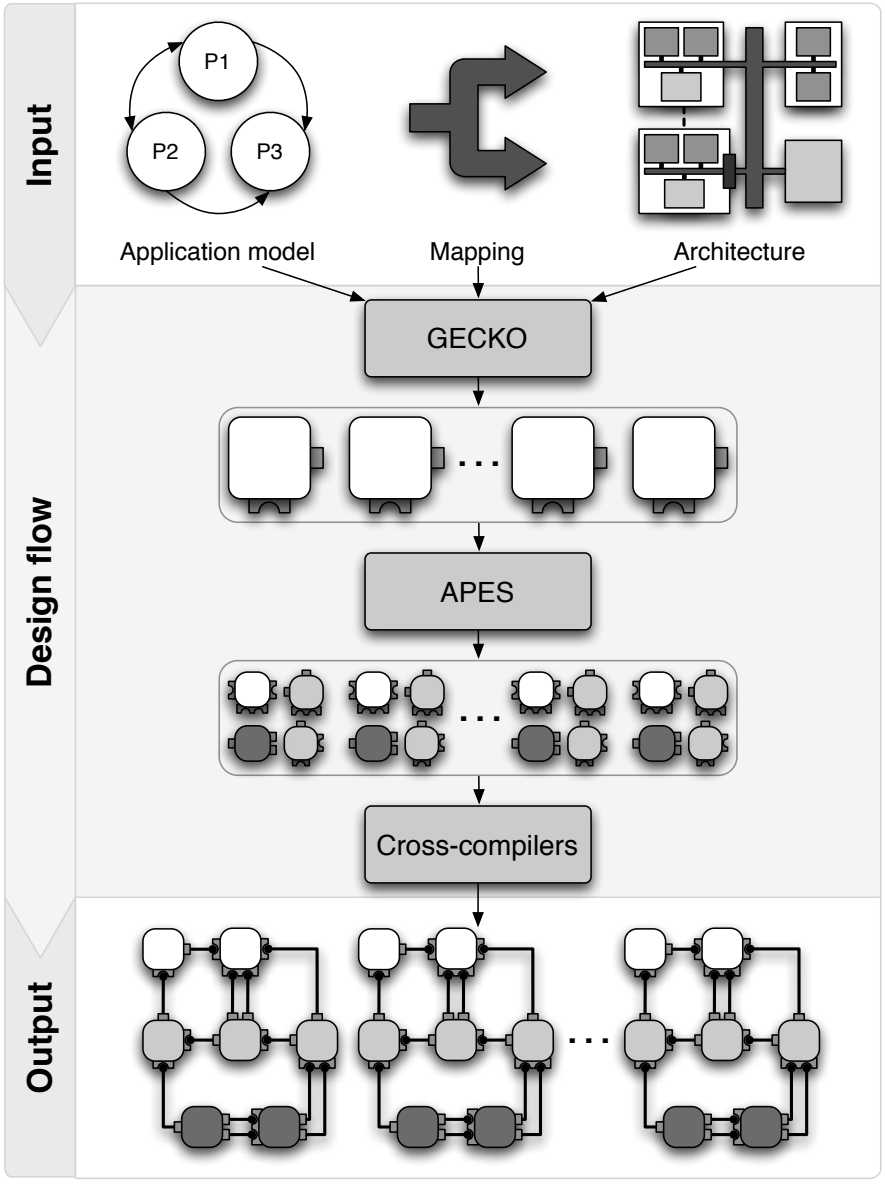


Figure 3.3: Embedded software design flow.

Based on this paradigm, the software development approach we propose in this work can be depicted as a flow composed of two consecutive steps (figure 3.3): the application code generation (GECKO) and the software component selection (APES). The application code generation starts from a model of the application and its mapping on the target hardware and generates the corresponding application components. The software component selection uses the application components to select their dependencies and generate their dependency graphs, which are in turn processed by the cross-compilers.

The output of this flow is a set of software binaries, one for each set of heterogeneous processors available on the hardware platform, each binary containing the implementation of the component dependency graph related to its piece of application. A more precise view of such a component dependency graph can be found in figure 3.4.

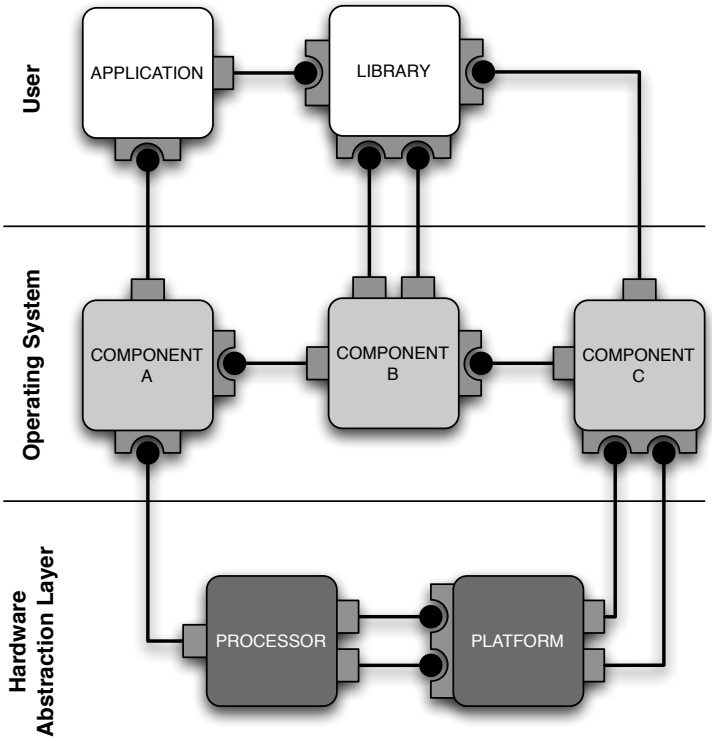


Figure 3.4: Implementation of a component dependency graph.

Its components are classified in three categories: user, operating system, and hardware abstraction. This classification enables the portability and scalability criteria and is detailed in chapter 4. The sections that follow detail both the application generation and the software component selection.

3.1 Application code generation

The application code generation uses elements passed as input — the application model, its mapping, and the target architecture — to generate the adequate number of software components corresponding to the application, as well as their implementations. These inputs must respect specific semantics since the code generator, Générateur de Code (GECKO), has to transform them into its own internal representation in order to perform the steps that will lead to a valid generation. GECKO's internal representation, the semantic constraints it imposes on the inputs, and its *modus operandi* are detailed in the following subsections.

3.1.1 Intermediate representation

In order to correctly generate source codes for multiple processor sets, a specific structural hierarchy that organizes processors, processes, functions, and communications is required.

We designed an object model, based on the Hierarchical Task Graph (HTG) [30] representation, that implements this hierarchy (figure 3.5) [36]. It defines three views: the platform view, the process network view, and the functional view. The construction of these different views can be achieved thanks to strong semantic constraints imposed on the three inputs files. These constraints are discussed in the next section. GECKO's intermediate representation is in many ways comparable to high-level models found in the works presented in section 2.4.3, and especially in [6, 13].

The platform view

The platform view represents the hardware architecture as it is described in the hardware architecture model. Only items of the type relevant to the code generation process, namely the processors (PROCESSOR on figure 3.5), are represented: depending on the brand and the class of the processor, important information concerning the type of compiler to use or the kind of optimization available can be deduced. In addition, processor representations are also containers that gather the processes to which they are dedicated.

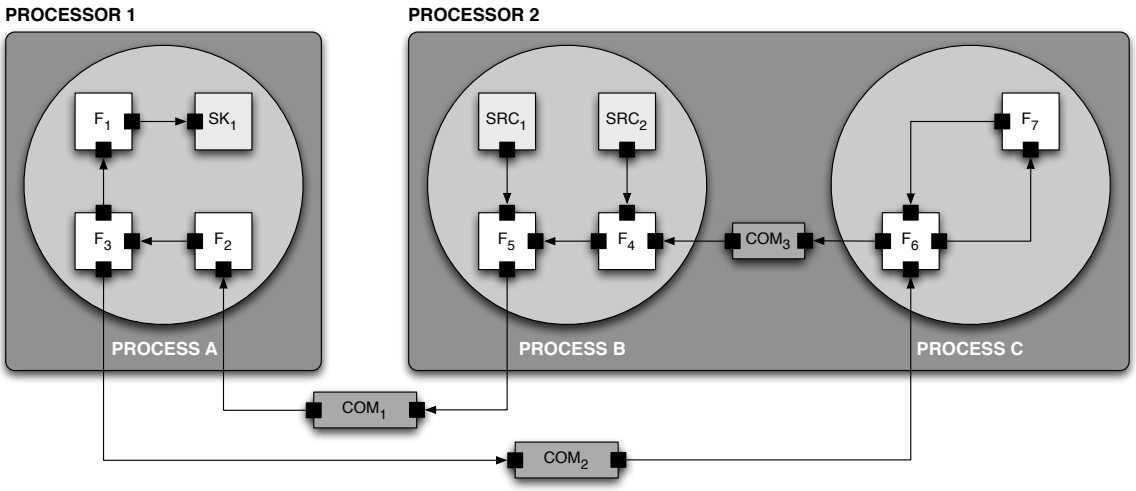


Figure 3.5: Representation of GECKO's intermediate model.

The process network view

The process network view represents the application's process network as it is described in the application model. Each process (PROCESS on figure 3.5) is an independent execution thread. **Communication between processes are explicit**: they must use communication channels (COM on figure 3.5) while implicit shared memory (usually by the mean of global variables) is prohibited. GECKO supports two kinds of processes: hierarchical processes that encapsulate functions graphs and virtual processes that merely point to source files.

The functional view

The functional view represents the application's function graph as it is represented in the application model. GECKO's functional view can faithfully describe any kind of CDFG [44]

providing that the graph contains discrete-time functions. **Communication between functions are implicit** and generally use local, shared memory. The code generator defines three classes of functional blocks: the sources, the functions, and the sinks (respectively SRC, F, and SNK on figure 3.5). These classes are later used by the generator's function scheduler in order to produce a source code functionally correct. It is important to note that the execution of the code generated from a function graph is correct with respect to the graph's data dependencies. However, our approach **does not guarantee** that the generated code will respect any specific execution properties embedded in the application model, such as the synchronous property in synchronous CDFG.

3.1.2 Semantic constraints

As presented in the previous subsection, each of the three views defined in GECKO's intermediate model is dependent on precise semantics that the input files must respect. The platform view relies on the abstract notion of processor not only as a configuration element that provides compilation information, but also as a container that regroups processes. Consequently, the hardware architecture model must define processors and the application's mapping must explicitly allocate processes to processors.

The process network view relies on the notion of independent processes and message-passing communications. These two elements are the foundation of *process networks* as they are defined in the *process algebra* approach (introduced in [39] and more generally presented in [5]). Simply put, GECKO is able to handle any kind of process network providing that it can be described using a process algebra, which is the case of most of the existing process network configurations. For instance, its representation model can describe a bounded KPN by using bounded FIFOs communication channels, or Communicating Sequential Processes (CSP) by using *rendez-vous* communication channels. As a consequence, the application model must contain a process network that can be described using a process algebra, supposedly valid.

The functional view relies on CDFGs containing discrete-time functions. Contrary to continuous-time functions, a discrete-time function is not continuously producing a result. Instead, its computation its triggered at periodical events on a time scale. These periodical events form the sampling period of a discrete-time function and its output is called a sample. In addition, a discrete-time function can have two behaviors. It can either be purely combinational — its inputs and outputs are available at the same execution cycle — or induce a delay: if its inputs are available at a time t, the function produces its outputs at a time $t + n$, with n the length of its delay.

The use of discrete-time CDFG is a sensible choice: any software algorithm can be implemented using this model of computation since software applications are executed by discrete-time machines. As a consequence, if the application model defines process behaviors they must be described as discrete-time CDFGs. These graphs should be free of any direct loops or feed-backs: this limitation is better explained in subsection 3.1.5. In addition, either the application model or the application's mapping must contain information concerning the association between processes and function groups.

3.1.3 *Modus operandi*

In order to generate the code for parallel processes from CDFGs, we opted for a mechanism similar to the simulation engine used in discrete-time software simulators such as Simulink

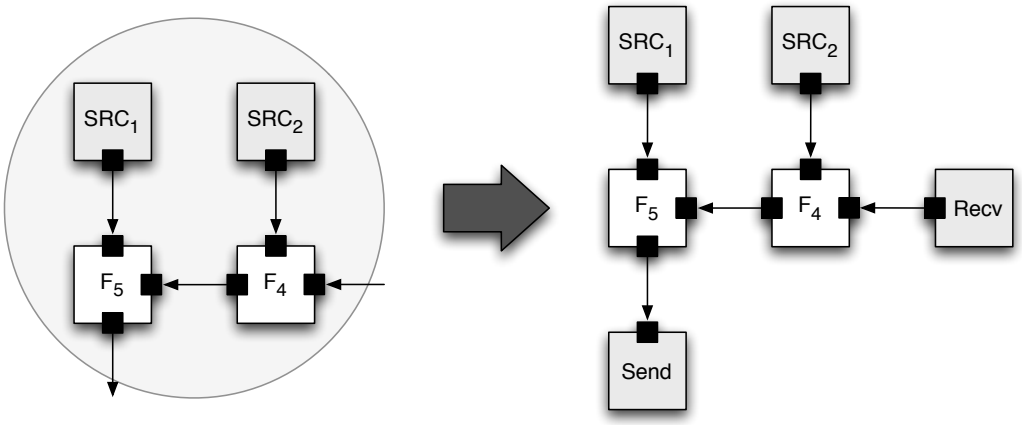


Figure 3.6: Expansion of external communication edges.

[10] or discrete-time hardware simulators such as VHDL [4] and SystemC [66]. Such an engine first finds the best sampling period for a given graph, then schedules its functions, and finally, for each simulation cycle, executes it as a whole. In our approach, each process repeatedly executes its own graph, which would have been statically scheduled beforehand by the code generator. To do so, it first parses the input files in order to create the corresponding intermediate model. Then, several operations such as external communications or delay-inducing functions need to be expanded in order to ensure the consistency of the function graphs. Finally, GECKO schedules the intra-process functions, optimizes the process data memory, and generates the software components.

3.1.4 Block expansion

Since an independent source code file has to be generated for each application process containing a function graph, these processes have to be scanned in order to check whether or not their function graphs are self-sufficient. For a function graph to be self sufficient, two properties must be verified: 1) each of its edges must be connected to two of its function and 2) delayed operations must be resolved.

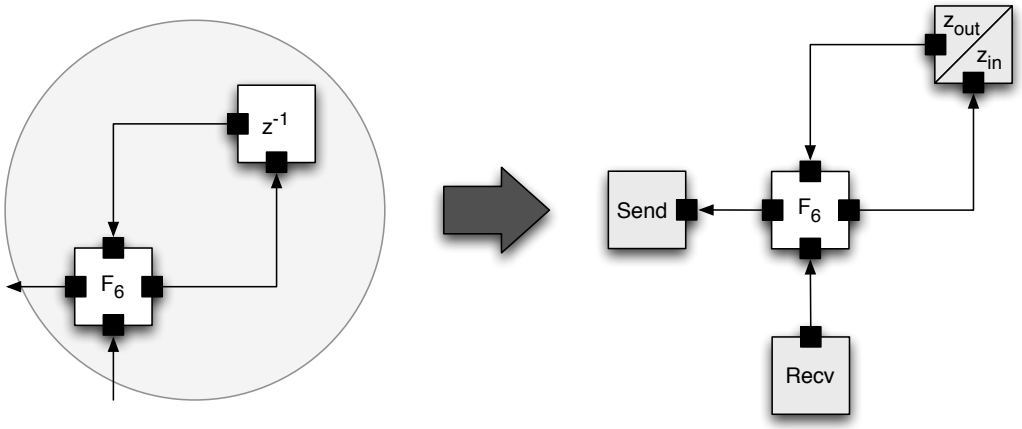


Figure 3.7: Expansion of delay blocks (z^{-1}).

The verification of the first property fails when an edge connects a function from one process' graph to a function from another process' graph. This is the case in inter-process communications. Indeed, according to the process algebra, direct shared memory is prohibited, which means that outgoing data transfer must be funneled through a message-passing communication block. As a consequence, the external link is replaced by an internal link to either a *Send* or *Receive* block depending on the direction of the communication, as depicted in figure 3.6.

The verification of the second property fails when one or more delay blocks are present. A delay block delays a discrete-time input by one or several samples depending on its configuration. In other words, a delay block acts as a data source and a data sink at the same time. In order to generate a valid code, a delay block is considered as both a source and a sink sharing the same memory point.

3.1.5 Function scheduling

GECKO implements a topological sort to schedule the processes' function graphs. A topological sort of a Direct Acyclic Graph (DAG) is a linear ordering of its nodes in which each node is ordered before all nodes to which it has outbound edges. In our case, the function graphs must be acyclic but can contain data feed-backs if they are constrained by a delay function. A stabilization mechanism is used to detect the presence of cycles.

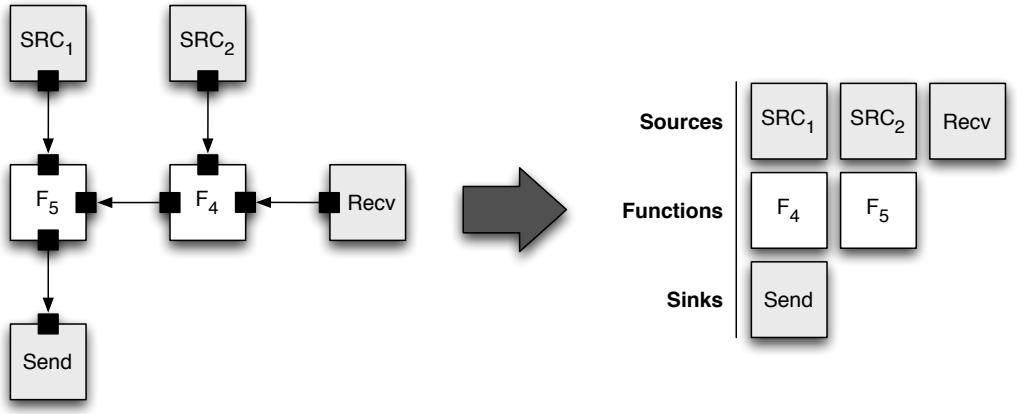


Figure 3.8: Construction of the scheduled block list.

At the end of the algorithm, a scheduled block list is created for each process contained in the intermediate representation (figure 3.8). This list is organized in function of the data dependencies: first the source blocks, then the functions blocks, and finally the sink blocks.

3.1.6 Data memory optimization

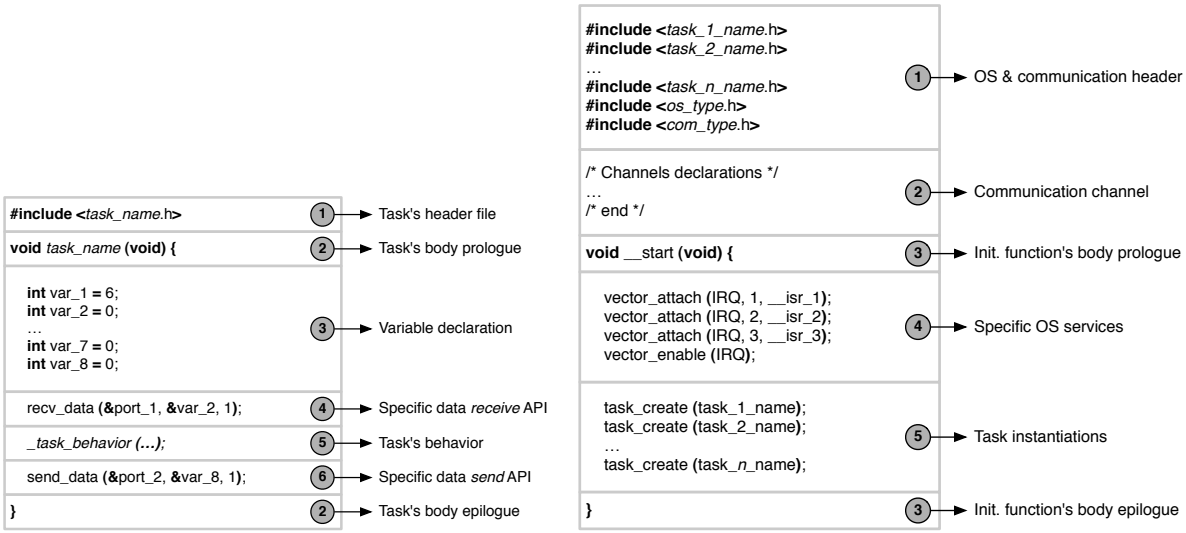
Two optimizations have been implemented in our code generation algorithm: the buffer copy removal and the buffer sharing [19, 36]. These optimizations are necessary since, by default, the memory allocator dedicates a memory space for each input and output of each function composing a task. The buffer copy removal optimization changes this behavior by allocating only one memory space per input/output couple (when possible) while respecting the data dependencies. Moreover, it happens that, during the execution of a task, at any given instant, certain memory spaces are no longer used. The buffer sharing optimization applies a liveness analysis on the execution list so as to find which memory space can be reused.

3.1.7 Software component generation

This operation includes three generation stages: the application's source code, the component's interface descriptor, and the linker command file.

Source code

GECKO produces the code for each process of each processor declared in the intermediate model (figure 3.9(a)). If the process does not contain a function graph and instead points to a source file, the source file is simply copied. First, it produces the necessary header commands and the process body prologue (figure 3.9(a), 1 and 2). Then, it parses the scheduled block list. The variable definitions are generated first, followed by the code of source blocks, function blocks, and sink blocks (figure 3.9(a), 3, 4, 5, and 6). Finally, the process body epilogue is generated (figure 3.9(a), 2). Concerning the *Send* and *Recv* blocks: an API parameter can be associated with the communication block to which they are connected. The code generator uses this parameter to produce the desired function calls.



(a) Source file (b) Initialization file

Figure 3.9: Example of generated files.

GECKO also produces an initialization file for each processor set defined in the intermediate model. It first generates the necessary header commands (figure 3.9(b), 1). Then, it scans the external communication blocks and generates the necessary channel declarations, according to the desired communication schemes (figure 3.9(b), 2). Next, the start function's prologue, specific OS's primitives related to communications, and process instantiations are generated (figure 3.9(b), 3, 4, and 5). In the same manner as communication blocks, an API parameter can be associated to processes and later used by the generator to produce the desired function calls. Finally, the start function's epilogue is generated (figure 3.9(b), 3).

Interface descriptor

In order to make the generated source code recognizable as a software component by the APES environment, GECKO needs to generates its interface descriptor file that contains a

summary of the provided and requested services present in the source code. Section 3.2.3 gives a complete description of this interface descriptor files.

Linker command file

A linker command file is a configuration file processed at link time by the object file linker specific to the target processor. Besides the general sections and memory configuration, several component-specific configuration parameters are declared in this command file (example 3.1, sections *.os_config* and *.hal*).

Example 3.1 Example of a linker command file for the GNU linker.

```
1  MEMORY
2  {
3    memory :  ORIGIN = 0x20000000, LENGTH = 0x0FFFFFFF
4  }
5
6  SECTIONS
7  {
8    .text 0x20000000 : { *(.text*) } > memory :text
9    .data ALIGN(0x8) : { *(.data*) } > memory :data
10   .bss  ALIGN(0x8) : { *(.bss*)  } > memory :bss
11
12   .os_config ALIGN(0x8): {
13     OS_N_DRIVERS     = .; LONG(0x1)
14     OS_DRIVERS_LIST = .; LONG(a_driver)
15     OS_KERNEL_HEAP_SIZE = .; LONG(0x10000)
16   } > memory :data
17
18   .hal ALIGN(0x8): {
19     CPU_OS_ENTRY_POINT = an_entry_point;
20     CPU_SVC_STACK = ABSOLUTE(ADDR(.sysstack))
21     PLATFORM_XXX_BASE  = .; LONG(0xFFFFEE00)
22   } > memory :data
23
24   .sysstack . + 0x10000: {} > memory
25 }
```

As a consequence, GECKO needs to produce one linker command file for each generated application component. Such a file is typically dependent of the compilation tool-chain. For instance, the GNU tool-chain uses *ldscript* [31].

3.2 Software component management

Although CBD improves the overall reuse of software components, it is still too coarse-grained to be completely satisfactory. For instance, let us consider a scheduler component with the following methods: `elect`, `switch`, and `suspend`. Let us suppose we have an implementation of a FIFO scheduler and we want to implement a round-robin scheduler. Theoretically, we would only have to refer to the FIFO scheduler and override the `elect` method. Technically however, we would have to duplicate the entire FIFO component to implement our round-robin component. With this kind of granularity, the CBD paradigm is not usable *as it is*: the bigger the code grows the less maintainable it gets. Another paradigm, called Object-Oriented Programming (OOP), offers a better granularity.

3.2.1 Object-Oriented Programming

OOP is a programming paradigm that uses "objects" – data structures consisting of data fields and methods – and their interactions to design applications and computer programs. Deborah J. Armstrong identified from nearly fourty years of computing literature some of its fundamental concepts [3]:

Class: A class does several things: at runtime it provides a description of how objects behave in response to messages; during development it provides an interface for the programmer to interact with the definition of objects; in a running system it is a source of new objects [78]. Based on these definitions, a class is: a description of the organization and actions shared by one or more similar objects.

Object: An object is a data carrier that can execute actions [78]. It also has been defined simply as something that has state, behavior, and identity [11], and as an identifiable item, either real or abstract, with a well-defined role in the problem domain [64]. By far the most common reference to an object is as an instance of a class [11]. Based on these definitions, an object is: an individual, identifiable item, either real or abstract, which contains data about itself.

Method: The concept of a method typically involves accessing, setting, or manipulating the object's data [64]. It is the fundamental element of an object. Based on these definitions, a method is: a way to access, set or manipulate an object's information.

Inheritance: Inheritance has been defined as a mechanism by which object implementations can be organized to share descriptions [93]. Another conceptualization of inheritance is as a relation between classes that allows for the definition and implementation of one class to be based on that of other existing classes [84]. Inheritance has also been explained (in association with the class hierarchy concept) as a mechanism by which lower levels of the hierarchy contain more specific instances of the abstract concepts at the top of the hierarchy [64]. Based on these definitions, inheritance is: a mechanism that allows the data and behavior of one class to be included in or used as the basis for another class.

Encapsulation: There are three primary conceptualizations of encapsulation in the reviewed literature. The first conceptualization of encapsulation is as a process used to package data with the functions that act on the data [93]. The second, most common conceptualization of encapsulation, is that encapsulation hides the details of the object's implementation so that clients access the object only via its defined external interface [11, 93]. The third conceptualization includes both of the previous definitions and can be summarized as: information about an object, how that information is processed, kept strictly together, and separated from everything else [64]. Bringing these conceptualizations together, encapsulation is: a technique for designing classes and objects that restricts access to the data and behavior by defining a limited set of messages that an object of that class can receive.

Polymorphism: The most basic conceptualization of polymorphism appears to be the ability to hide different implementations behind a common interface [95]. Different other conceptualizations can be found in the reviewed literature [50, 64, 84]. Bringing them together, polymorphism is defined as: the ability of different classes to respond to the same message and each implement the method appropriately.

Compared to the CBD, OOP ensures a better flexibility and a finer grained reuse of the code. However, OOP heavily relies on complex, performance consuming, runtime mechanisms often not adapted to resource-constrained hardware.

3.2.2 Literature review

Encapsulating software pieces in order to improve the flexibility, portability, and scalability of a software project is not a new endeavor. This is especially true in the field of operating systems, or more precisely operating system kernels. What follows summarizes the most notable efforts made in this domain.

Some systems such as Amoeba [87], Mach [1], or SPIN [9] already tried to encapsulate functionalities by moving some of their functions into user-level processes called servers, thus breaking with monolithic designs like UNIX [77]. Referred to as first-generation μ-kernels, they were still too monolithic and showed poor performance compared to their ancestors. Other systems such as Windows NT [83] or eCos [58] define a set of hardware-dependent functions, called HAL, to be used throughout the kernel. Thus, kernel functions are written only once since the porting only requires rewriting the HAL for the target processor. The time required for this operation closely depends on the number of functions it contains.

Second-generation μ-kernels such as Exokernel [23] or L4 [52] propose radically new approaches where all the protocol-related functions are implemented as servers. Only mandatory mechanisms are kept in the kernel (address space, inter-process communication and basic scheduling) allowing this generation of kernel to fare better than the previous one. By design, these μ-kernels are rather modular. Servers can be added or removed without modifying the kernel. They can be started and stopped at will, and only the servers required by the application are mandatory.

While these μ-kernels also have been used to build pure object-oriented solutions [8,16,62,63, 75], they suffer from two main drawbacks: 1) each piece of the kernel must be written specifically for the target processor, inducing non-negligible porting times; 2) servers are full-blown processes relying on the address space mechanism of the kernel, which require a hardware Memory Management Unit (MMU). Unfortunately, this kind of hardware is seldom provided on heterogeneous MP-SoCs, making pure μ-kernel design not suitable as a generic solution for these platforms.

Pure component-based approaches such as OSKit [25], Peeble [26], and Think [24] specify the basic operating system services and organize them into components. They usually use an Interface Definition Language to select the components required by the application and resolve their dependencies. With their strict design they take advantage of an increased flexibility, but they also suffer from a coarse-grained vision of their software component. As a consequence, these systems are often hard to port and to maintain.

Last but not least, the Aspect-Oriented Programming (AOP) approach, when coupled to a well-defined separation of concerns, enables fine-grained compositions of system-level functions [55, 56, 81]. However, it intensively relies on aspect extensions (such as AspectC++ [57] or AspectJ [45]), which are not unified across programming languages and only exist for a few of them (such as Java, C++, and C in its embryonic from).

3.2.3 Object-Component Model

An ideal solution would be to combine the lightweight approach of CBD with the fine-grained code reuse offered by OOP. This proposition is not so preposterous given that we are not inter-

ested in the dynamic properties of these approaches, which are one of the reason an external runtime is usually required. Indeed, the static characteristics of these two approaches are theoretically close: CBD's interface/implementation couple can be assimilated to an OOP class, a CBD instance can be assimilated to an OOP object, and OOP's encapsulation can be assimilated to CBD's interface mechanism and its public/private categories. Consequently, it is possible to bring fine-grained code reuse to CBD, **through the inheritance and overriding polymorphism mechanisms**.

We felt the necessity to redefine a paradigm for the following reasons: there is no noted authority in the IDL topic; inheritance and polymorphism are not supported in any IDL; we don't want to restrain the programmer to a particular programming language; Object-Oriented Programming Language (OOPL) and CBD rely on runtime mechanisms, generally not compatible with resource-constrained hardware platforms[1]. Instead, we prefer to use a Model-Driven Engineering (MDE) approach to define components[2]. MDE is a software development methodology which focuses on creating models, or abstractions, closer to some particular domain concepts rather than computing (or algorithmic) concepts. It aims at maximizing compatibility between systems and simplifying the design process [80].

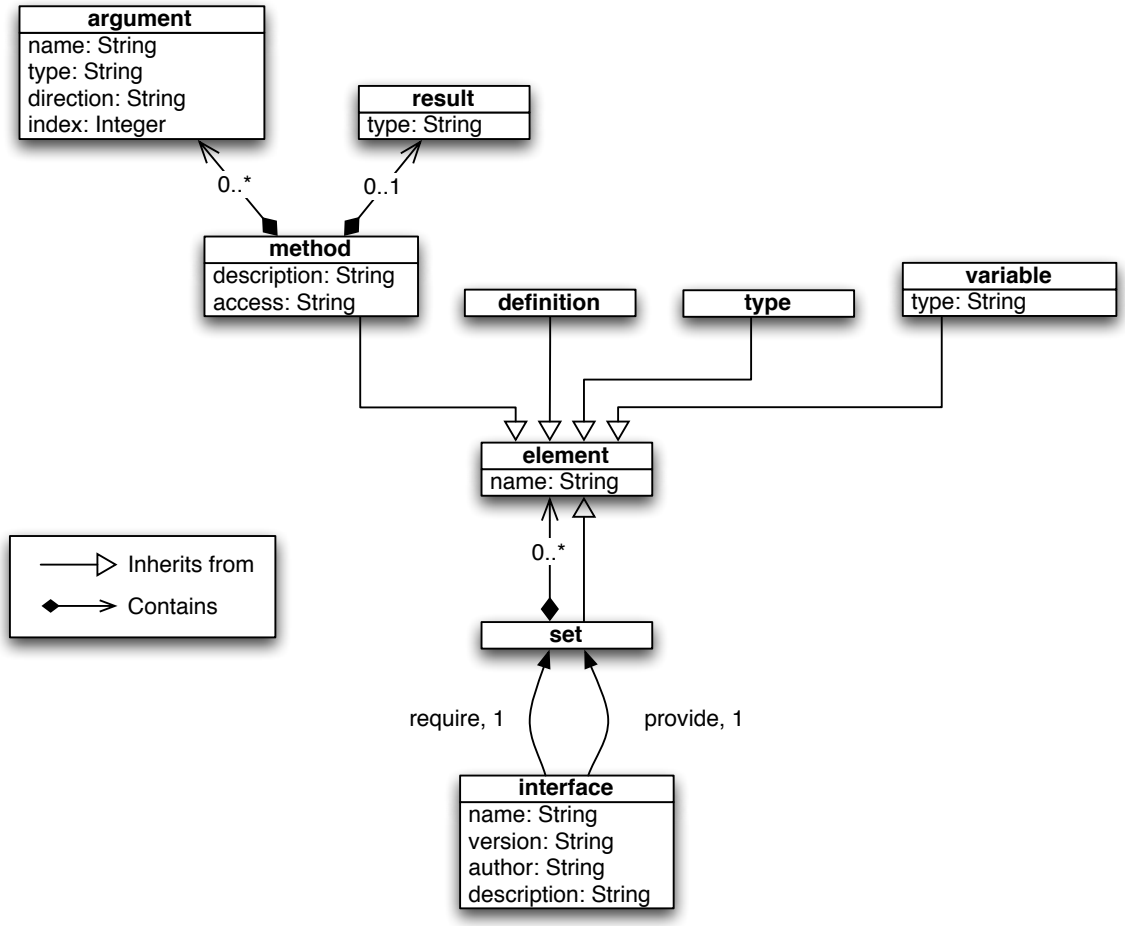


Figure 3.10: Class diagram of a component's interface meta-model.

[1]One could argue that ADA is used to develop embedded applications [49]. ADA is an exception to the rule since its inner construction allow dramatic optimizations of the generated code. But even choosing ADA wouldn't have been a solution flexible enough, since we did not want to be restricted to a particular programming language.

[2]This work has been done in collaboration with Guillaume Godet-Bar and Amin Elmrabti.

In the case of our paradigm, named Object-Component Model (OCM), MDE allows the use of encapsulation and some object-oriented mechanisms without using extra software mechanisms or enforcing a particular programming language. To do so, each component has one Interface Descriptor (IFD) and one Implementation Descriptor (IMD). The Interface Descriptor (IFD) is used to describe the component's interface, its functional dependences, and generate the related files in a predefined programming language. The IMD is used to describe the component's implementation. For both descriptors, we defined their construction rules (comparable to a grammar), called meta-model. Identically, the IFD is called an interface model and the IMD is called an implementation model.

Interface descriptor

Several notions needed to be defined in order to enable thorough interface descriptions: *definition*, *type*, *variable*, *method*, *argument*, and *result*. Most of these notions are common to imperative programming languages and are self-explanatory. Figure 3.10 gives the complete representation of the interface meta-model we defined. The *type* and the *definition* are defined as singletons representing their name. The *variable* is defined as a duplet that contains its name and its type. An argument is defined as a quadruplet composed of its name, its type, its direction, and its index. A method is defined as a triplet that contains its name, its set of arguments (which can be empty), and its result. It also contains information such as a description and an access privilege (*public* or *private*).

Example 3.2 Interface of basic scheduler component (scheduler.xmi).

```
1  <interface author="XavierGuerin" name="Scheduler" version="1.0" unique="true">
2    <provide> <!-- Provided section -->
3      <context name="Operations">
4        <method name="scheduler_elect" description="Elect a thread">
5          <result type="thread_t"/>
6        </method>
7        <method name="scheduler_switch" description="Switch between two threads">
8          <argument name="old" type="thread_t" direction="out"/>
9          <argument name="new" type="thread_t" direction="in"/>
10       </method>
11     </context>
12   </provide>
13
14   <require> <!-- Required section -->
15     <type name="thread_t"/>
16   </require>
17 </interface>
```

We also integrated a grouping mechanism to enable the management of the inter-component dependencies. IFD contains two groups, *provide* and *require*, that in turn can contain *definitions*, *types*, *variables*, and *methods* that the component either provides to the rest of the world or requires to operate. We chose XML Metadata Interchange (XMI) as a persistent representation of IFD. An example of IFD for a basic scheduler component is in figure 3.2

Implementation descriptor

For each element defined in the IFD, its implementation must be present in the corresponding IMD. The nature of the code integration into an IMD has yet to be decided, even though

solutions such as directly embedding the code or using a remote repository have been investigated. The main challenge involved in the design of the Implementation Descriptor was the introduction of the inheritance and polymorphism mechanisms. The goal of this operation was not to be fully compliant with OOP, but to reduce the amount of code duplication in order to enhance the maintenance and the reuse of software components.

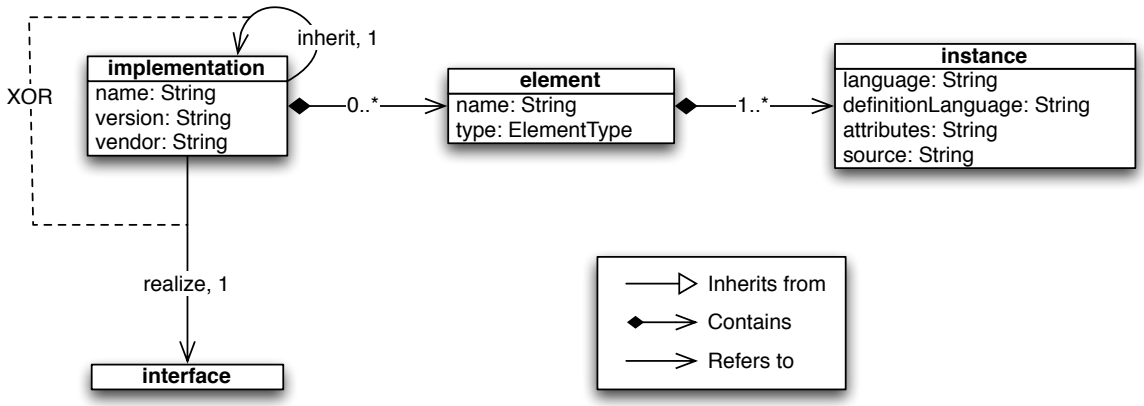


Figure 3.11: Class diagram of a component's implementation meta-model.

Therefore, we limited ourselves to single inheritance and overriding polymorphism. This subset is quite convenient since we were able to reach our goal while being compatible with virtually any kind of imperative programming language. An example of IMD for a First In - First Out (FIFO) scheduler component is given in example 3.3.

Example 3.3 Implementation of a FIFO scheduler component (scheduler_fifo.xmi).

```
1  <implementation author="XavierGuerin" name="SchedulerFIFO" version="1.0">
2    <realize>
3      <interface href="scheduler.xmi"/>
4      <element name="scheduler_elect" type="Method">
5        <implementation language="C" source="scheduler_elect_fifo.c"/>
6      </element>
7      <element name="scheduler_switch" type="Method">
8        <implementation language="C" source="scheduler_switch.c"/>
9      </element>
10   </realize>
11 </implementation>
```

To enable the single inheritance mechanism, we included in the implementation meta-model the ability to either *realize* an interface or *inherit* from an other implementation (cf example 3.11). When the implementation *realizes* an interface, then each entry in this interface must be implemented. However, when an implementation *inherits* from an other implementation, then it inherits all its realizations and each redefined entry overrides the one of its parent.

An example of simple inheritance and overriding polymorphism is given in example 3.4. The implementation of the scheduler component given in example 3.3 does not support priorities in its method `scheduler_elect`. With OCM, it is easy to inherit from this FIFO scheduler (using the `from` tag) and override the `scheduler_elect` method with a new one (example 3.4).

Example 3.4 Inheritance and overriding of example 3.3: FIFO scheduler with priority.

```
<implementation author="XavierGuerin" name="SchedulerFIFOPrio" version="1.0">
  <inherit>
    <from href="scheduler_fifo.xmi"/>
    <element name="scheduler_elect.c" type="Method">
      <implementation language="C" source="scheduler_elect_fifo_prio.c"/>
    </element>
  </inherit>
</implementation>
```

3.2.4 Relation with the source code

Since OCM does not impact the actual source code, the relations between the content of an IFD and its corresponding implementation need to be explained. Items present in the *require* section of a component are simply being used in its source code. For instance, a required type is used to declare a variable, or a required method is actually called from the component's implementation. Example 3.5 shows the use of three required elements (a type, a definition, and a method) in a C source file.

Example 3.5 Required elements in a C source file.

```
/* Use of a required type and a required definition */
spinlock_t lock = DEFAULT_LOCK_VALUE;

/* Use of a required method */
lock_acquire (lock);
```

Items present in the *provide* section of a component are declared either in the public header files of the private header files of the component's implementation (e.g. the C `'.h'` files or the ADA `'.ads'` files). Example 3.6 shows the declaration of three provided elements in a C header file.

Example 3.6 Provided elements in a C header file.

```
/* Declaration of a definition */
#define DEFAULT_LOCK_VALUE 0

/* Declaration of a type */
typedef uint32_t spinlock_t;

/* Declaration of a method */
extern void lock_acquire (spinlock_t lock);
```

We used the C language to write our examples, but any other language compatible with this interface/implementation separation and able to bind with external objects could have been used instead. Such languages are (and not limited to) C++, C#, Objective-C, Pascal, ADA, Modula, Oberon, Eiffel, as well as any kind of assembly, providing that the compiler used supports this mechanism. This is mainly the case of the GNU Compiler Collection (GCC) included in the GNU compilation toolchain.

3.3 Graph construction engine

The previous section explained how the concepts included in OCM efficiently enable the reuse and the maintenance of existing pieces of software code. This section shows how the CBD origins of OCM also enable the automation of the component graph construction and as well as the automatic compilation of its corresponding binary object.

3.3.1 Theory

An MDE approach has to be rather attractive when it is proposed to a world of C and assembly code programmers. This is what we kept in mind when we designed OCM: the concepts we gathered in our development approach not only alleviate the pain of code maintenance and reuse but also enables the automation of final binary constructions. This last operation can be achieved thanks to the dependency information included in each interface descriptor.

Solutions such as Think [24] already addresses this issue with the help of *composition descriptor files*. A composition descriptor file declares which software component to use for a given configuration, thereby forming a full instance of the configuration's final component graph. Although this solution allows to automatically build the application's binary, it still obliges the software developer not only to be aware of all the components his application requires (and the components required by the components required by its application, etc.) but also to actually know their identifiers. In order to completely automate this process, we developed an algorithm that relies on both the functional dependencies defined in OCM's interface descriptor and the full set of available components. This set, called $\mathbb{C}$, is considered a 4-partite digraph (figure 3.12).

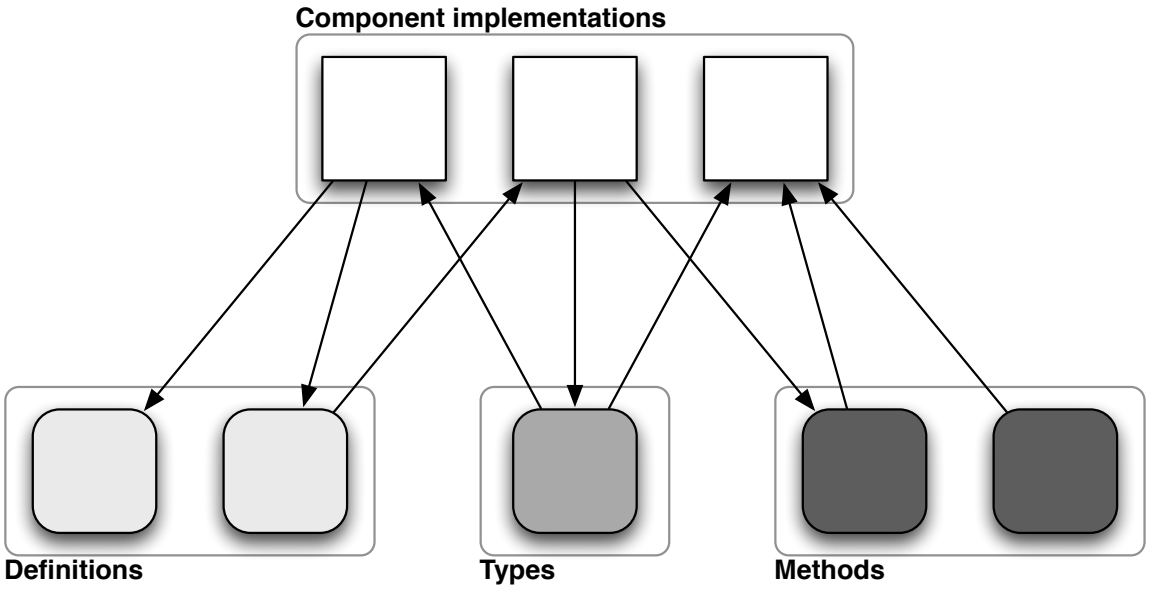


Figure 3.12: 4-partite digraph of the complete component set.

A k-partite graph is a graph whose vertices can be partitioned into k disjoint sets so that no two vertices within the same set are adjacent. $\mathbb{C}$ is partitioned into four disjoint sets: the component implementation set, the type set, the definition set, and the method set. The

component implementation set contains the implementations of the available software components. The type, definition, and method sets respectively contain all the types, definitions, and methods available across the full set of components. An arc starting from an element of the component implementation set and pointing to an element of one of the other sets represents a *require* relation from the component element to the other element. An arc starting from an element of one of the type, definition, or method sets and pointing to an element of the component implementation set represents a *provide* relation from the component element to the other element.

Algorithm 1 Basic graph construction algorithm.

1. **function resolve** $(\mathcal{C}, \mathbb{D}, \mathbb{C})$ **is**
2.
3. $\mathbb{D}' \leftarrow \emptyset, \mathbb{D}_{final} \leftarrow \emptyset, \mathbb{T} \leftarrow \emptyset, \mathbb{G} \leftarrow \emptyset, \mathbb{M} \leftarrow \emptyset$
4.
5. {Match $\mathcal{C}$'s requirements with each element of $\mathbb{C}$}
6. **for all** $\mathcal{C}' \in \mathbb{C}$ **do**
7. **if** $\mathcal{C}_{types} \subseteq \mathcal{C}'_{types}$ **then**
8. $\mathbb{T} \leftarrow \mathbb{T} \cup \{\mathcal{C}'\}$
9. **end if**
10.
11. **if** $\mathcal{C}_{definitions} \subseteq \mathcal{C}'_{definitions}$ **then**
12. $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{C}'\}$
13. **end if**
14.
15. **if** $\mathcal{C}_{methods} \subseteq \mathcal{C}'_{methods}$ **then**
16. $\mathbb{M} \leftarrow \mathbb{M} \cup \{\mathcal{C}'\}$
17. **end if**
18. **end for**
19.
20. {Check if at least one dependency exists for each set}
21. **if** $\mathbb{T} = \emptyset$ **or** $\mathbb{G} = \emptyset$ **or** $\mathbb{M} = \emptyset$ **then**
22. **abort** "Unresolved dependency."
23. **end if**
24.
25. $\mathbb{D}' \leftarrow (\mathbb{T} \cup \mathbb{G} \cup \mathbb{M}) \setminus \mathbb{D}$ {prevent redundancy}
26. $\mathbb{D}_{final} \leftarrow \mathbb{D}'$
27.
28. {Apply this algorithm to each dependency}
29. **for all** $\mathcal{C}' \in \mathbb{D}'$ **do**
30. $\mathbb{D}_{final} \leftarrow \mathbb{D}_{final} \cup$ **call resolve** $(\mathcal{C}', \mathbb{D}', \mathbb{C})$
31. **end for**
32.
33. **return** $\mathbb{D}_{final}$

It could have been possible to partition the initial set into only two disjoint subsets regrouping, for the one, the component implementation and, for the other, all the types, definitions, and methods. In this configuration, dependencies would be resolved from the component implementation set relatively to the $type \cup definition \cup method$ set. This is too coarse a grain since it could lead to falsely resolved dependencies and would eventually produce invalid

dependency graphs. Indeed, if a given component does not provide a particular type to another component but do provide a method required by this other component, the dependency would be resolved for both the required type and the required method.

3.3.2 Graph construction algorithm and production of the binary

Our construction algorithm starts from three elements: 1) a root component called $\mathcal{C}$; 2) a dependency set called $\mathbb{D}$ and initially empty; and 3) the full set of components $\mathbb{C}$. Then, it computes $\mathcal{C}$'s dependencies and recursively analyzes them, as detailed in algorithm 1. However, the graph produced by this algorithm is not final yet. Although it reduces the initial component set and takes care of unresolved dependencies (which correspond to colored vertices with no outbound arcs in figure 3.12), it cannot deal with dependence conflicts (which correspond to colored vertices with multiple outbound arcs and at least one inbound arc in figure 3.12).

One solution to this issue is to add one more parameter and one more return value to the algorithm, which correspond to a set of conflicting components with an initial value of $\emptyset$. If the returned set is not empty, the caller matches the conflicting components with those of another set, called *restriction set*, containing the user's selection in case of conflict. A good example of conflicting components are HAL components. Indeed, all of them are able to resolve a dependency with one of the HAL interface's element. As a consequence, the user must provide the algorithm with his preference in terms of HAL components.

Algorithm 2 Graph compilation algorithm.

```
1.  function build (D_final) is
2.
3.  D_compiled ← ∅
4.
5.  {Parse D_final and compile its elements}
6.  for all C ∈ D_final do
7.          D_compiled ← D_compiled ∪ call compiler (C)
8.  end for
9.
10. {Produce the final binary}
11. call linker (D_compiled)
12.
13. return
```

Once the component graph has been constructed, another algorithm iteratively parses each component of the $\mathbb{D}_{final}$ set, building its objects using an adequate compiler. Last but not least, the final application binary is produced by calling an adequate linker on the $\mathbb{D}_{compiled}$ set (algorithm 2).

3.4 Summary

In this chapter, we presented a novel approach to develop embedded software targeting MPSoC architectures. In combining both an application code generator and software component selection mechanism, our software design flow reconciles strong design properties in

an efficient way. We also showed that Component-based Development (CBD) and Object-Oriented Programming (OOP) are both powerful mechanisms for the development of embedded software applications but that, taken alone, none of them are sufficient to meet the constraints inherent to embedded platforms. We detailed the Object-Component Model (OCM), a clever mix of those mechanisms that takes advantage of their theoretical compatibility to bring high levels of flexibility and automation to embedded application programming.

4
Software framework

The complexity of heterogeneous MP-SoCs brings a new challenge when designing a software environment targeting this kind of architecture. Like their brethren supporting uni- or heteroneneous, multiprocessor embedded systems, they need to be small and efficient. But they also need to be portable and scalable in order to make the application development on MP-SoC platforms completely seamless. The previous chapter presented our embedded software design flow making use of Object-Component Model (OCM) to provide high levels of flexibility and automation.

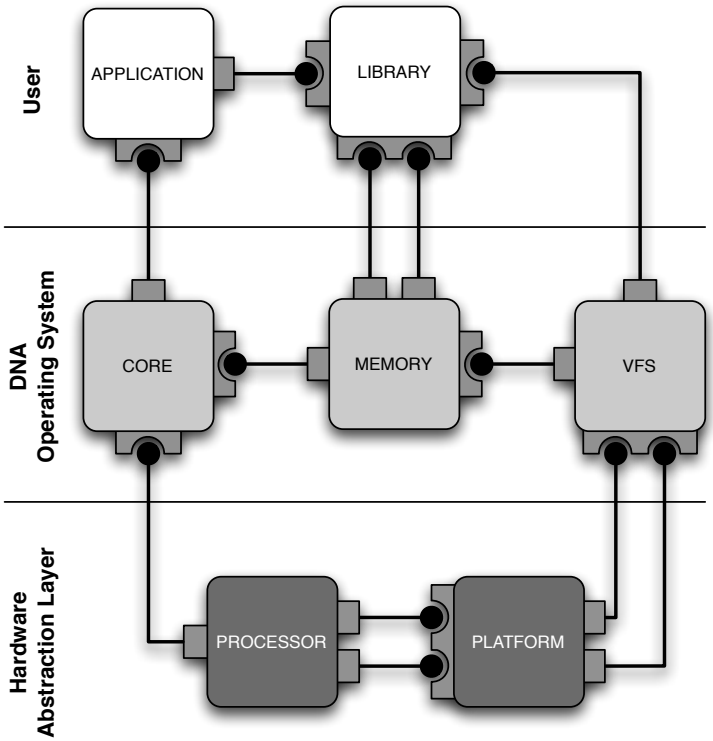


Figure 4.1: Categories of the software framework.

This chapter presents the software framework we developed around OCM to propose high levels of portability and scalability. This framework is composed of **various software compo-**

nents organized in three categories (figure 4.1): user application and libraries, the DNA-OS, and the Hardware Abstraction Layer. The following sections detail these three categories. Concerning the user components, we only focus on the libraries present in the framework.

4.1 Hardware abstraction layer

The Hardware Abstraction Layer provides the only access to hardware-specific primitives. It aims at abstracting the functionalities of an "ideal platform" and an "ideal processor". Its design has been greatly inspired by the eCos's HAL [58], but revised for heterogeneous MP-SoCs and simplified in order to reduce its implementation overhead: it exports all in all 27 functions, where eCos's HAL amounts to 50 functions and the Linux's *arch* interface to over a hundred. It has been split into two OCM components: the platform component and the processor component.

4.1.1 Platform component

From a hardware point of view, a platform refers to hardware devices organized around communication infrastructures. From a software point of view however, the platform can be seen as a super-entity that can deliver information that would otherwise be hard-coded in the software, such as the global endianness and the global multiprocessor configuration. The role of a platform component is to provide this mechanism.

Global endianness

Two processors of a heterogeneous MP-SoC may not share the same endianness. Hence, data exchanged between applications running on these processors need to be decoded back and forth. This operation is usually done at the application level, where the endianness differences are handled either directly by the application (with tools such as External Data Representation (XDR) [86]) or by a specific communication API (as in Message-Passing Interface (MPI) [60]). These solutions were developed at a time when applications were distributed in networks of machines running different operating systems and required a unified programming interface.

When the same operating system is used, this operation can be executed at the communication-driver level where the data can be decoded in the `read` and the `write` functions. At this level, the remote endianness would have to be hard-coded since only raw data is processed here, making the driver dependent on a specific hardware configuration. This issue has been solved by defining an endianness shared by all interprocessor communications. For each exchanged data the driver calls on one of the endianness-related processor primitives that relies on this value to decide whether or not the data needs to be converted. More importantly, only one version of the driver is necessary since these primitives are available for any kind of processor.

Global multiprocessor configuration

The global multiprocessor configuration contains information about all the SMP clusters present on heterogeneous MP-SoCs. They are used by the processor component to get values such as the number of processors of a given type.

Device description headers

The platform components also embed program headers containing the functional organization of its hardware devices. These descriptions can be used later either by device drivers or directly by the application, through the processor's `read`/`write` functions. Gathering device descriptions in a single component limits the possible duplications of such description throughout the software framework, therefore reducing the proliferation of potential bugs.

4.1.2 Processor component

From a core integer unit, modern processors have evolved into more complex entities (containing a MMU, instruction and data caches, floating-point units, lock mechanisms, etc.) making them difficult to characterize in a generic manner. This statement is even more true with embedded processors since most of them can be now configured at will. The role of a processor component is to cover most of the properties and functions that modern processors share, namely: the endianness, multiprocessor configurations, I/Os, execution context, synchronization, traps, power management, and memory and caches.

Endianness

The endianness management interface contains data flow primitives, data exchange primitives, and data construction primitives.

- The **data flow** primitives are useful when the endianness of data is known by the programmer and may (or may not) differ according to the processor.
- The **data exchange** primitives are useful to deal with data shared by several processors on the same heterogeneous MP-SoC. They use the global endianness to decide whether or not a conversion is necessary.
- The **data construction** primitives construct/split a bigger data type from/into smaller entities (e.g. constructing a 64-bit word from two 32-bit words, splitting a 32-bit word into two 16-bit words). They are useful for adapting data between two different specifications.

These functions are to be used wherever the endianness may be problematic. This is usually the case in device drivers (when the endianness understood by a device is different from the processor's endianness), file system modules (e.g. the endianness of the FAT format [61]), or in data flow applications (e.g. the endianness of the data in a MJPEG flow [74]).

Multiprocessor configurations

The multiprocessor management interface contains interprocessor synchronization primitives and a processor count primitive.

- The **interprocessor synchronization** primitives block and resume the extra processors. They are useful in certain areas of a multiprocessor kernel, which are required to be run only once (especially at boot time).

- The **processor count** primitive returns the number of processors of the same type as the current one, using the global multiprocessor configuration. In traditional operating systems, the number of manageable processors is usually hard-coded in the kernel. This value is used to initialize, at boot time, several structures that contain miscellaneous information about each processor in the system. To increase the portability of the kernel this value is often set to a high bound, inducing unnecessary memory overheads.

This kind of solution is suboptimal on systems such as heterogeneous MP-SoCs where the memory resources are scarce. The processor count primitive directly offers the correct value. The main benefit is that the kernel doesn't have to be reconfigured when the number of processors differ from one platform to the other. This system is very helpful in simulation and architecture exploration environments such as [27] and [73], which two of the main parameters are precisely the number and the type of processors.

Inputs/Outputs

The I/O management interface abstracts the different `read` and `write` operations of the processor. It contains default operation primitives, non-cached operation primitives, and vector operation primitives.

- The **simple** primitives read or write a data without any particular restriction. They must be able to reach any location registered in the address map. For most of the processors these primitives are implemented using their `load` and `store` mnemonics. In the case of deeply embedded processors, such as the mAgicV processor in the Atmel D940 board, they are implemented using its tightly coupled Direct Memory Access (DMA) engine since only its internal memories can be accessed with its `load` and `store` mnemonics.
- The **non-cached operation** primitives read or write a data without using the data caches. For processors without data caches, cached operations and non-cached operations are identical. Some processor offer specific instructions, such as SPARC's `lda` and `sta`, to perform non-cached operations. For the others, simple operations are used since cachability is an attribute of their memory maps.
- The **vector operation** primitives read or write a sequence of data. These primitives are usually implemented with a loop of default operations, except when the processor is coupled with a DMA engine or a vector co-processor.

These methods are usually useful to write device-specific, operating system drivers: when widely used, the resulting implementation of the driver can be completely generic regarding the processor architecture.

Execution context

The execution context management interface defines primitives that can be used to load, store, initialize, and switch between execution contexts. It also defines an opaque type for the processor's context. This interface can be used by operating system schedulers or application level context operations such as `setjmp` and `longjmp` of the C library. For most of the processors, these primitives are implemented using successions of `load` and `store` mnemonics (e.g. for the ARM processors, the specific `ldm` and `stm` mnemonics are used). For some DSP

processors, these primitives are irrelevant since no solution has yet been found to perform an efficient context switch on a processor architecture that can contain $256 \times 40 - bit$ registers or private memories of $x \times y$ bytes.

Synchronization

The synchronization management interface defines primitives that can be used to perform *test-and-set* and *compare-and-swap* operations. For most of the processors, they are implemented using their respective atomic operations (`swap` for the ARMs, `ll/sc` for the MIPS, etc.). Concerning DSP processors or some kinds of uCs, these operations are implemented without guarantee of the atomic property since they usually don't have any atomic operation. Instead, these processors are provided with hardware lock engines that can be used by device drivers to guarantee the protection of a critical section.

Traps

The traps management interface contains primitives to manipulate hardware interrupts and processor exceptions:

- The **interrupt** primitives propose to attach a handler to a software or a hardware interrupt, and enable/disable the hardware interrupts. In addition, this interface defines opaque types for the state of the interrupts, the interrupt handlers, and the exception handlers that can be used with its methods.
- The **exception** primitive attaches an exception handler to one of the processor's exception vector. In addition, this interface defines opaque types for the exception handlers and the exception identifiers.

The implementation of these functions may vary from one processor to another. For most of the processors, the handlers are stored in vector tables. For some embedded RISC processors or uCs, the interrupt handlers are directly stored in the registers of their dedicated Advanced Interrupt Controller (AIC), while the exception handlers are stored in a vector table. For some DSP, the handlers are stored in specific registers.

Power

The power management interface contains primitives that control the power consumption of the processor. In the current state of the framework, this point has not yet been fully covered. However, it already contains a primitive to enter a *wake-on-interrupt* mode if available, and will certainly contain methods to control the processor's operating frequency and voltage.

Memory and caches

The memory and caches management interface defines primitives that can be used to manipulate the processor's memory management unit (if any) and caches (if any). In the current state of the framework, this point has not yet been fully covered. However, concerning the MMU management, it will most certainly define an opaque type that describes the MMU's

page table, methods to map and unmap pages into/from a page table, and grant pages from a page table to another. Concerning the cache management, it already contains a method to flush the write buffer and in the future will certainly define methods to flush a line/all of the data/instruction cache into the memory, and fetch a line/all of the data/instruction cache from the memory.

4.2 The DNA operating system

DNA-OS is not just another operating system. When we designed our OCM-based software organization, we realized that we needed a system kernel compatible with our HAL, with a small impact on the memory footprint (under 32 kilobytes) and ideally the application's performance, a sufficient amount of advanced features in order to support the most widespread application libraries such as a fully fledged C library or a PThreads library, and an advance level of compatibility with different processor architectures (RISC, DSP, uC, ...). In addition, it should be component-based and compatible with OCM.

Considering all this prerequisites, it would have been rather difficult to adapt an existing solution (from either one of the mainstream solutions or a solution form the literature) to our needs. For this reason, we built the DNA-OS using a component-based design inspired by the μ-kernel-like architecture of the Be Operating System [34] and a strong separation of functional concerns as emphasized in [62, 81], in which we injected the support of the HAL. As a consequence, the implementations of the DNA-OS components are 100% generic.

4.2.1 Core component

The Core component is in charge of the processor's core functions. Its interface contains thread management methods, scheduling methods, spin-lock operations, semaphore management methods, messaging capabilities, alarm management methods, and interrupt management methods. This component is the base of the kernel and is the next entity to start after the bootstrap. It makes use of the execution context, traps, synchronization, and multiprocessor interfaces of the HAL.

Threads

The execution thread (or simply thread) is the basic execution unit of the DNA-OS kernel. The default kernel/user ratio is $1:1$ (one user thread for one kernel thread). The $1:N$ ratio is eventually supported, although left at the application developer's discretion. The choice of supporting only an $1:1$ ratio in the kernel has been made to take advantage of multicore systems, since it is the only ratio that enables fine-grained thread migration between homogeneous processors. Therefore, a thread can freely migrate from processor to processor or be pinned on a specific processor.

The Core component provides the standard, self-explanatory thread operations: `create`, `destroy`, `suspend`, `resume`, `wait`, `yield`, and `exit`. It also provides three non-standard operations: `snooze` suspends a thread for a specified amount of time, `find` retrieves a thread identification number from its name, and `get_info` gets information about a thread. Each of these functions makes use of the context interface of the HAL.

Scheduling

The scheduling part of the Core component is responsible for the execution of threads on available processors. It provides five internal operations: `elect` that elects the next available thread, `dispatch` that dispatches a thread on a processor, `push` that pushes a recently freed processor into the pool of processors, `pop` that pops a free processor from the pool of processors, and `switch` that switches between two threads.

This set of operations has been so carefully designed as to be easily derivable using the OCM inheritance mechanism. The reader can refer to example 3.4 for an example of inheritance concerning the `elect` function. In its current version, the Core component is able to schedule any number of threads on any number of processors, on either uniform or non-uniform architectures. Each of these functions makes use of the context interface of the HAL.

Semaphores

The Core component provides a system-wide semaphore mechanism. Its interface contains functions to `create`, `destroy`, `acquire`, and `release` semaphores. A timeout can be specified to acquire and release methods. By default, the release method reschedules the calling thread if another one is waiting to acquire the semaphore. A specific flag can be used to prevent this behavior. Each of these functions makes use of the multiprocessor, synchronization, and memory interfaces of the HAL.

Messaging

The Core component provides a fast inter-thread communication mechanism as introduced in the first μ-kernel designs. Threads accepting messages ask the kernel for the creation of named ports that could later be used by other threads to send messages. Its interface contains methods to `create` and `destroy` ports, `create`, `destroy`, `send`, and `receive` a message, and is compatible with the optimizations described by Jochen Liedtke in [51]. These functions make use of the multiprocessor, synchronization, and memory interfaces of the HAL.

Alarms

The Core component provides also a time framework based on the concept of alarm. This framework supposes that each processor present on the hardware has its own timer (as it is usually the case on Uni-Processor (UP) and SMP systems). These timers are then used to program kernel and user alarms. On the kernel side, these alarms are used to manage timeouts and gather execution information. The available operations are `set_alarm`, `reset_alarm`, and `cancel_alarm`. The current version of the component implements the time framework in a *tickless* fashion.

Interrupts

While the HAL component deals with the low-level trap management, the Core component provides a higher level of abstraction that allows, for instance, the management of multiplexed interrupts. It provides two operations: `attach` that attaches an interrupt to a specific interrupt line on a particular processor and `detach` as an inverse operation. The current

implementation of this mechanism bypasses the demultiplexer when it is not required (e.g. when only one ISR is attached to an interrupt line). This optimization enables the management of complete cases without penalizing the performance on simple cases. Each of these functions makes use of the trap interface of the HAL.

4.2.2 Virtual File System Component

The Virtual File System (VFS) component is in charge of the high-level I/O operations. It exports an interface compatible with the *directories and files* interface and the *advanced file management* interface of the POSIX standard [40] such as `read`, `write`, `mkdir`, `opendir`, and so on. Contrary to second-generation μ-kernels, it is present in the kernel to ensure high performance as well as compatibility with processors deprived of hardware MMU.

This component also manages the file system components. It makes use of the multiprocessor, synchronization, and I/O interfaces of the HAL. Two versions of this component currently exist: a full implementation, which is able to mount file systems, manage file system components, and deal with any kind of files or directories, and a namespace-based implementation limited to accessing devices.

4.2.3 Memory Component

The memory component provides dynamic memory services at the kernel- and user-levels. It has no functional dependencies with other components and can be easily removed from configurations that do not require a dynamic management of the memory. The kernel-level interface contains two methods to allocate memory and free previously allocated memory in the kernel space. The user-level interface (including memory pages management) is still a work in progress, but already contains a basic support of the POSIX's `sbrk`.

In the future, advanced page management and `sbrk` local to TCMs will be added. It makes use of the multiprocessor, synchronization, and memory interfaces of the HAL. Only one version of this component is currently available. It implements a *worst-fit* allocator for the kernel-level interface and a basic `sbrk` implementation for the user-level interface.

4.2.4 Modules

There is three types of modules understood by the DNA-OS's components: file system components, device driver components, and extension components. Each of these components are presented in the following subsections.

File System Components

The file system components manipulate files on specific file systems. They are managed by the VFS component. Each file system component exports a set of methods corresponding to most of the POSIX file manipulation function. They are used when one of the VFS operations is executed on a file managed by the file system component. Two default file systems are provided: `rootfs` and `devfs`. `rootfs` is in charge of populating `'/'` and allows DNA-OS to boot without an external root file system. `devfs` manages the device drivers. It populates itself by calling some methods of the device driver component interface as described below. File system components usually make use of the multiprocessor, synchronization, and I/O interfaces of the HAL.

Device Driver Components

The device driver components manipulate hardware devices. They are managed by the `devfs` file system component. Each device driver component exports the names as well as a subset of the POSIX file manipulation functions (basically `read`, `write` and `ioctl`) corresponding to each device class it manages. The names and subsets of these functions thus exported are used by the `devfs` as file entries. Each call of `read`, `write`, or `ioctl` on one of the `devfs`'s file is thereby rerouted to the device driver's own functions. Device driver components make use of the multiprocessor, synchronization, and I/O interfaces of the HAL.

Extension Components

The extension components can be seen as kernel-level libraries. They gather functions that other kernel components may share. Most extensions have their own interfaces that, contrary to the other components of the framework, are not normalized. Such components can be bus managers, partition scheme decoders, runtime compatibility layers, etc.

4.3 User libraries

User libraries contain functions relying on the operating system that the application can directly use. They can take various forms: language support libraries, thread libraries, network libraries, and so on. In this section, we present the user libraries proposed in our framework: a POSIX-compliant implementation of the POSIX threads, a implementation of the Distributed Operation Layer (DOL) interface, the Newlib C library, and a port of the LwIP TCP/IP stack.

4.3.1 POSIX threads support

PThreads [40] is a POSIX standard for threads. It defines an API for creating and manipulating threads. PThreads are most commonly used on Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris, but Microsoft Windows implementations also exist. The PThreads component present in our framework is solely based on DNA-OS's API. Thread and synchronization operations are fully supported and attribute manipulation operations are only partially supported.

4.3.2 Distributed Operation Layer interface

DOL is a framework that enables the (semi-) automatic mapping of applications onto the multiprocessor SHAPES architecture platform. The DOL consists of basically three parts [88]:

DOL Application Programming Interface: The DOL defines a set of computation and communication routines that enable the programming of distributed, parallel applications for the SHAPES platform. Using these routines, application programmers can write programs without having detailed knowledge about the underlying architecture.

DOL Functional Simulation: To provide programmers a possibility to test their applications, a functional simulation framework has been developed. Besides functional verification of applications, this framework is used to obtain performance parameters at the application level.

DOL Mapping Optimization: The goal of the DOL mapping optimization is to compute a set of optimal mappings of an application onto the SHAPES architecture platform. In a first step, XML based specification formats that allow to describe the application and the architecture at an abstract level have been defined. Still, all the information necessary to obtain accurate performance estimates are contained.

The DOL component present in our framework offers a full implementation of the DOL API. It has been used in the SHAPES [68] European project. It is important to note that the DOL formalism is also a supported format in our design flow [17].

4.3.3 Redhat's Newlib C library

Newlib [76] is a C library intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses, which make them easily usable on embedded products. Newlib is only available in source form. It can be compiled for a wide array of processors and will usually work on any architecture with the addition of a few low-level routines. Newlib is present in our framework as a virtual component, meaning that only the functional dependencies are described. The reason is that Newlib is directly integrated with the compilation tool-chain. We implemented most of the Newlib's OS-dependent system calls. The only exception is process management, since DNA-OS does not yet support processes.

4.3.4 LwIP TCP/IP network stack

Lightweight IP (LwIP) is a light-weight implementation of the TCP/IP protocol suite that was originally written by Adam Dunkels of the Swedish Institute of Computer Science [20] but now is being actively developed by a team of developers distributed world-wide headed by Leon Woestenberg. Since it is release, LwIP has spurred a lot of interest and is today being used in many commercial products. LwIP has been ported to several platforms and operating systems and can be run either with or without an underlying OS. The focus of the LwIP TCP/IP implementation is to reduce the RAM usage while still having a full scale TCP. This makes LwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

4.4 Use cases and tools

Depending on the application's needs, the proposed software framework can be used to generate multiple configuration. In this section, we present three examples: a minimal configuration that only relies on the hardware abstraction components, a small configuration that relies only on a part of the operating system, and a large configuration that relies on the operating system and external compatibility libraries.

4.4.1 Minimal configuration

The application of the minimal configuration relies only on the services of the hardware abstractions components. Figure 4.2 presents the component graph of such an application. It

features two kinds of components: the application itself, in light gray, and the HAL components (platform and processor) in dark gray.

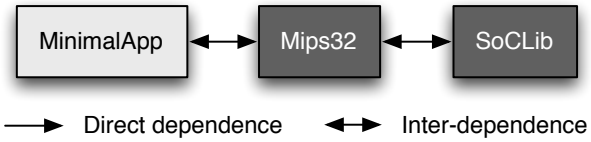


Figure 4.2: Component dependency graph of a minimal configuration.

This kind of configuration can be useful on hardware platforms with a highly constrained memory capacity. Depending on the application — more precisely depending on how many hardware abstraction functions are used in the application — the framework's overhead may vary from a few hundreds bytes to approximatively 4 KB. It can also help to build very small pieces of software such as ROM-based *boot loaders*.

4.4.2 Small configuration

The application of the small configuration relies on the Core component of the DNA-OS operating system to provide high-level thread and multiprocessor services. Figure 4.3 presents the component graph of such an application. It features the three kinds of components: the application, the HAL, and the operating system components in medium gray.

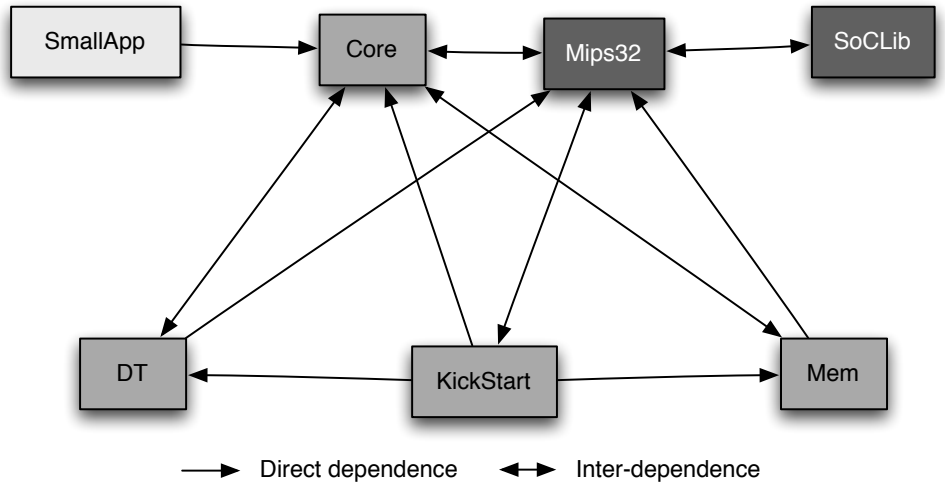


Figure 4.3: Component dependency graph of a small configuration.

The operating system components are: the core component (Core) that provides multithreading, the memory component (Mem) that provides dynamic memory, the *kickstart* component that boots the operating system, and the DNA tools component (DT) that provides some tools to the operating system. This kind of configuration can be useful on massively parallel systems such as the picoChip's picoArray [67] or Tilera's tiled architecture [92]. Also depending on the application, the framework's overhead may vary from a few kilobytes (with the multithreading only) to 24 KB (with multithreading, alarms, semaphores, and messaging).

4.4.3 Large configuration

The application of the large configuration relies on every component of the operating system as well as several external libraries, several device drivers and file systems, and several user libraries. Figure 4.4 presents the component graph of such an application. It features the three kinds of components: the HAL, the operating system components in medium gray, the application, and the user libraries.

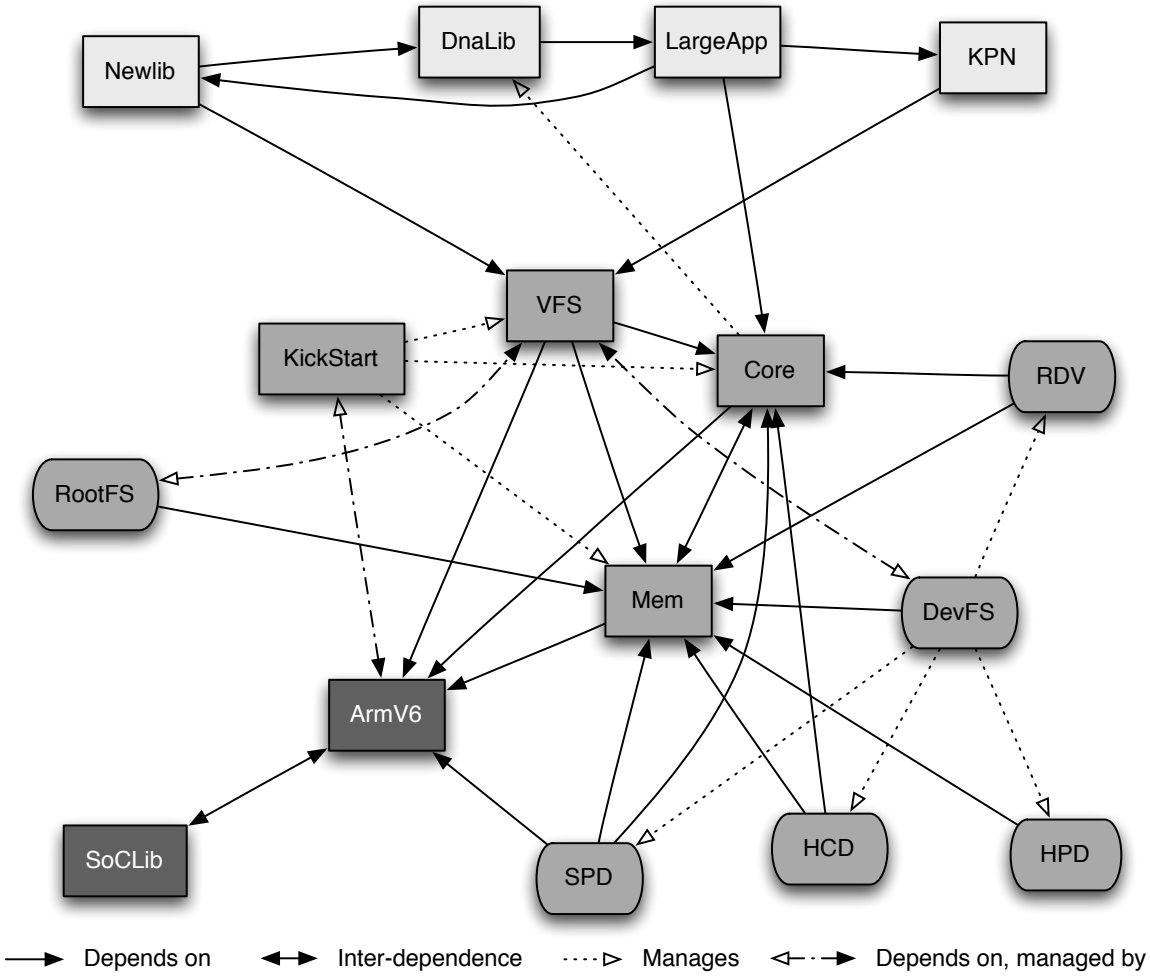


Figure 4.4: Component dependency graph of a large configuration.

The user libraries are: the DNA library component (DnaLib) that provides easy access to DNA-OS's system calls, the Kahn process network component (KPN) that provides communication channels, and the Newlib C library component (NewlibC). The operating system components are: the Core component (Core), the VFS that provides file management, the Memory Manager component (Memory), the *kickstart* component, several device drivers (RDV, SPD, HCP, and HPD), and several file systems (RootFS and DevFS).

This kind of configuration is typically used for multithreaded, multiprocessor, multimedia applications (it is used for the MJPEG decoder application described in section 5.3.1). Depending on the application, the framework's overhead may vary from approximatively 20 KB up to 120 KB. This is mainly due to the NewlibC library that by itself weights 100 KB.

4.4.4 Tools

Along with software components, APES provides a set of tools that manipulates these components: a dependency analyzer, a dependency graph resolver, a component compiler, and a graph generator.

Dependency analyzer The APES dependency analyzer, **apes-depend**, analyzes the direct dependences of a given root component. It is mainly used by the dependency graph resolver.

Dependency graph resolver The APES dependency resolver, **apes-resolve**, resolves the dependences of a given component as described in section 3.3. To do so, it first loads the descriptions of the existing software components, then recursively analyzes the functional dependencies of the root component, deals with missing references and component conflicts, and finally generates a minimal set of components on which the root component depends.

Component compiler The APES component compiler, **apes-cc**, uses the dependency resolver to build a minimal set of software components for a given root component. Then, using the dependency analyzer, generates the correct compilation command for each source file of each component and executes it. Finally, it gathers all the produced object files and links them altogether.

Graph generator The APES graph generator, **apes-graph**, uses the dependency resolver to generate a serialized version of a given root component's dependency graph. The DOTTY [22] format is used to generate this file.

4.5 Summary

In this chapter, we presented a complete software framework that aims at enhancing the development of embedded software targeting the MP-SoC architecture. In using both the OCM paradigm and a strict separation between the hardware dependent software, the operating system, and the application, this framework brings high levels of portability and scalability to embedded software programming. The combination of this framework with the embedded software design flow presented in the previous chapter makes a strong, efficient software development approach that scores high marks for all the criteria.

5
Experimentations

The main goal of the work presented in this document is to provide an ideal programming environment to develop complex, parallel, embedded applications on heterogeneous MP-SoCs. To do so, we designed a software development approach that gathers the properties of flexibility, scalability, portability, and automation. In this chapter, we present two case studies where our approach has been used to port complex applications on three different platforms. First, section 5.1 refreshes the memory of our reader with a short description of our design flow and gives the conditions of its application for each experiment. Then, section 5.2 and 5.3 deal with the experiments themselves. Finally, section 5.4 reviews the current status of our software framework.

5.1 Conditions of experimentations

The proposed software development approach can be depicted as a flow composed of two consecutive steps (figure 5.1): the application code generation (GECKO) and the software component selection (APES). The application code generation starts from a model of the application and its mapping on the target hardware and generates the corresponding application components. The software component selection uses the application components to select their dependencies and generate their dependency graphs, which are in turn processed by the cross-compilers. The output of this flow is a set of software binaries, one for each set of heterogeneous processors available on the hardware platform and each binary containing the implementation of the component dependency graph related to its piece of application.

The first experiment proposes to port a complex, parallel application first on the Atmel D940 MP-SoC, then on the ShapOtto platform, starting from a high-level representations of the application, the hardware, and the mapping of the one onto the other. First, we describe the application and the hardware platforms. Then, we present the programming model proposed for the experiment as well as the operating strategy we chose to apply. Finally, we detail the two major steps of our design flow: 1) the generation of parts of the application code using GECKO, and 2) the production of the final application binaries using APES. It is important to note that an early version of APES has been used in this experiment, which was not fully automated. Therefore, the production of the binaries relies on a component configuration file integrated into the compilation environment and containing a description of the software component graph.

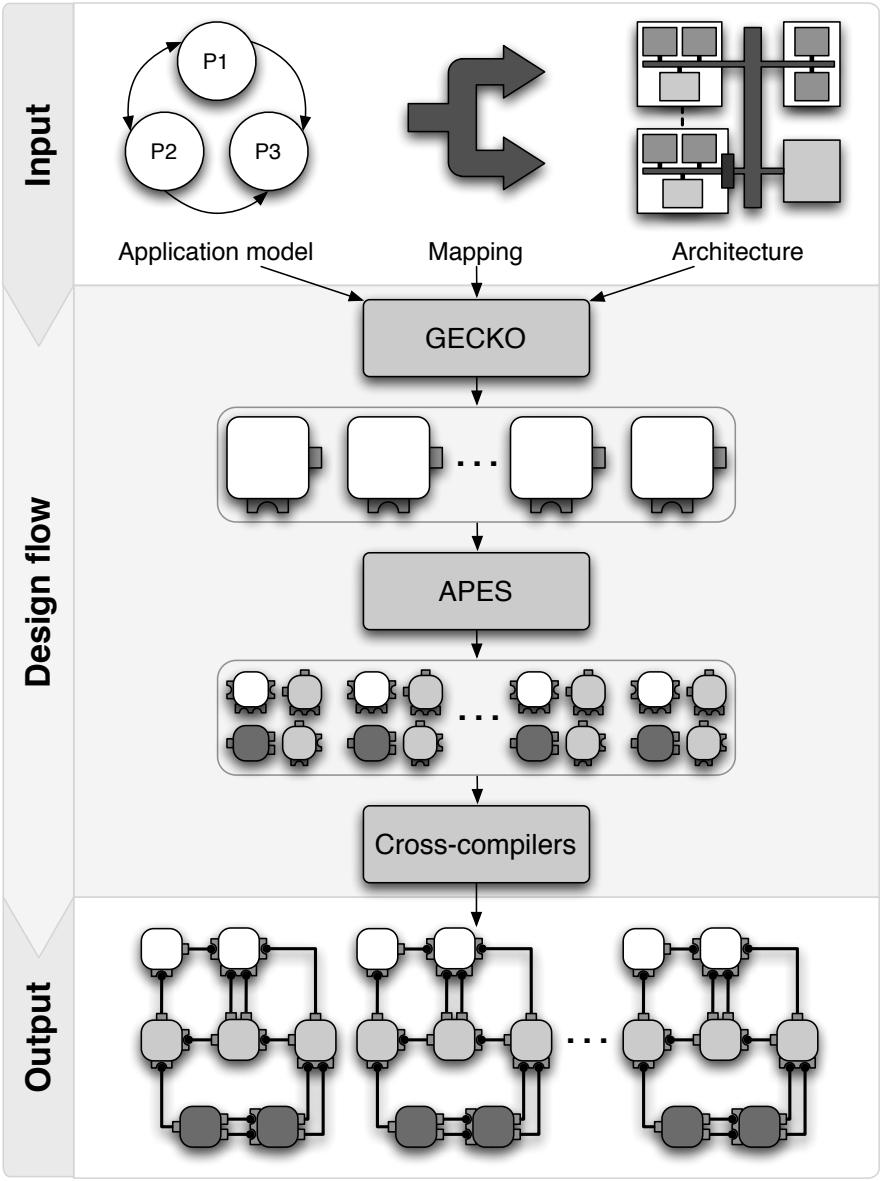


Figure 5.1: Embedded software design flow.

The second experiment proposes to port two multimedia applications on HeSyA, a heterogeneous, simulated MP-SoC, starting from preexisting source codes using common APIs. The organization of this experiment is close to the first one. First, we describe the applications and the hardware platform. Then, we present the programming model imposed by the applications. Finally, we detail execution of our software design flow. Since we started from existing applications we directly used APES to produce the application binaries. However, the version of APES used in this experiment is based on OCM and consequently features an automated construction of the software component graphs.

5.2 First experiment

The challenge of this experiment was to provide an automated design flow able to generate application binaries for both a heterogeneous and a distributed hardware architecture: the

Atmel D940 MP-SoC and its multi-tile evolution, the ShapOtto platform. It has been conducted in the scope of the SHAPES European project. In this section, we first present the principles of the application chosen for this experiment, the Lattice QCD (LQCD). Then, we present the two hardware architectures as well as their programming model. Next, we detail the operating strategy we chose to apply to execute the application on the two different platforms. Finally, we detail the execution of our design flow and we present several results.

5.2.1 LQCD application

The Quantum Chromo-Dynamics (QCD) is the regnant theory of strong interactions. It is formulated in terms of quarks and gluons which physicist believe are the basic degrees of freedom that make up hadronic matter. It has been very successful in predicting phenomena involving large momentum transfer. In this regime the coupling constant is small and the perturbation theory becomes a reliable tool. On the other hand, at the scale of the hadronic world, $\mu \precsim 1$ GeV, the coupling constant is of order unity and all perturbative methods fail. In this domain lattice QCD provides a non-pertubative tool for calculating the hadronic spectrum and the matrix elements of any operator within these hadronic states from first principles. LQCD can also be used to address issues like the mechanism for confinement and chiral symmetry breaking, the role of topology, and the equilibrium properties of QCD at finite temperature.

LQCD is QCD formulated on a discrete Euclidean space/time grid. and can be simulated on a computer using methods analogous to those used for Statistical Mechanics systems. These simulations allow to calculate correlation functions of hadronic operators and matrix elements of any operator between hadronic states in terms of the fundamental quark and gluon degrees of freedom [33].

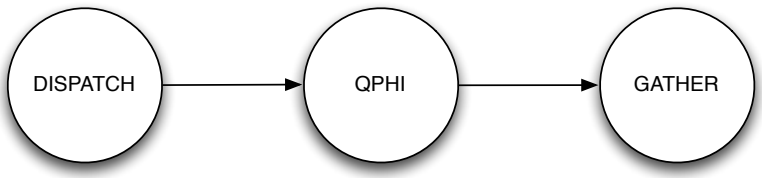


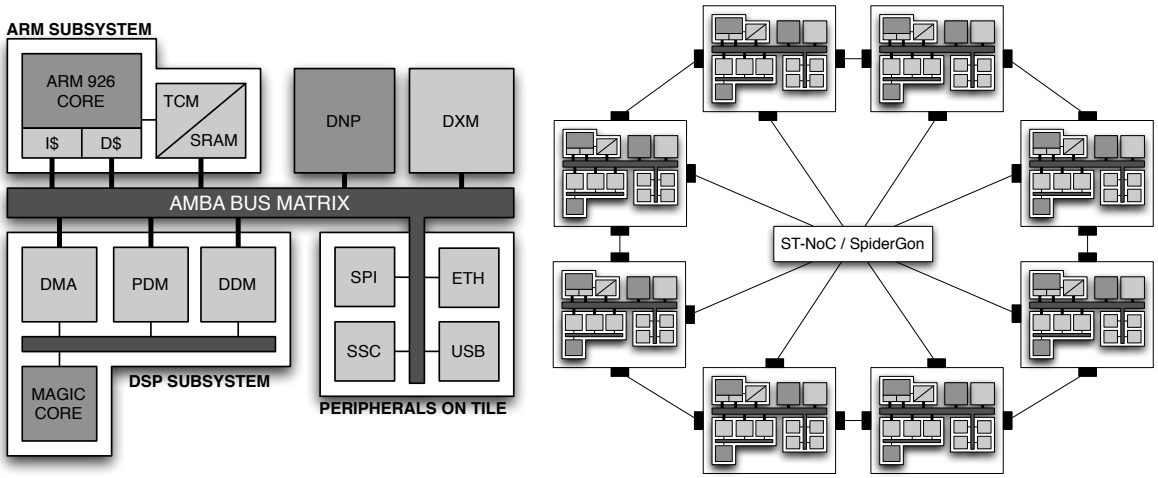
Figure 5.2: Basic LQCD processing unit.

The algorithm proposed by the Istituto Nazionale di Fisica Nucleare (INFN) has been developed for that purpose. It takes as input a complete lattice of $n \times n \times n \times n$ basic elements defined as spinors and its corresponding gauge matrices, and computes a Dirac operator on each of the spinors. This type of input data favors its processing parallelization: the initial (and usually large) lattice can be divided in smaller matrices which computation can be distributed over a cluster of computing unit. If necessary, these smaller matrices can slightly overlap each other to reduce potential border effects. In addition, the INFN people parallelized the process itself in three tasks (figure 5.2): the DISPATCHER task marshals a small matrix and funnels it to the QPHI process; the QPHI task applies the Dirac operator on the matrix; the GATHER task gathers the computed matrix from the QPHI task.

5.2.2 Atmel D940 SoC and the ShapOtto platform

The Atmel D940 is the last high performance MP-SoC from the Diopsis line. It contains three subsystems: an ARM subsystems, a DSP subsystems, and a peripheral subsystem (figure 5.3(a)). The ARM subsystem is composed of an ARM926EJ-S core running at 200 MHz and 48 kilobytes of local memory (SRAM). The ARM core can access its local memory either through the AMBA matrix or though its local Tightly Coupled Memory (TCM) bus. In the former case each word access costs 2 cycles, while in the latter each word access costs 1 cycle. Although the TCM mode is faster, it is also more restrictive since the available memory is divided as follows: $\frac{1}{3}$ reserved for the instructions, $\frac{1}{3}$ reserved for the data, and $\frac{1}{3}$ usable as SRAM, or $\frac{1}{2}$ reserved for the instructions and $\frac{1}{2}$ reserved for the data.

The DSP subsystem is composed of a MagicV DSP core running at 100 MHz, 8 kilowords of program memory (Program DSP Memory (PDM), 16 kilowords of data memory (Data DSP Memory (DDM)), a DMA controller, and a hardware semaphore controller. The MagicV is a 128-bit Very Long Instruction Word (VLIW) processor able to execute eight instructions per cycle for a maximum of perform 1.6 Giga Floating Point Operations Per Seconds (GFLOPS). Although the core itself can only access its own memory directly and needs its DMA controller to access other memories, the PDM and DDM are memory-mapped and can consequently be accessed by the ARM processor or some peripherals. In addition, the DMA controller and the hardware semaphore engine are memory-mapped. The D940 also contains several peripherals gathered in their own subsystem. Named Peripherals-On-Tile (POT), it contains various hardware devices such as an Universal Serial Bus (USB) controller, an ethernet controller, a Synchronous Serial Controller (SSC) controller, and so on.



(a) Atmel D940 SoC (b) ShapOtto platform

Figure 5.3: Hardware platforms of the SHAPES project.

A specific processor, named Distributed Network Processor (DNP), can also be found. Its purpose is to interact with a NoC and to enable the construction of a clustered architecture. This is the cornerstone of the ShapOtto platform (figure 5.3(b)), which joins eight D940 SoC in a tiled architecture, where each occurrence of a D940 is called "a tile". Each tile can communicate with the others through a Spidergon ST-NoC [18] driven by its DNP.

5.2.3 Programming model

One of the SHAPES project's objectives was to offer to application programmers an unified programming approach for every processor/tile of the target hardware platforms. The proposed approach was the Distributed Operation Layer (DOL) [88] developed by the Eidgenössische Technische Hochschule (ETH) in Zürich. A short presentation is given in section 4.3.2. In practice, a DOL application is composed of two sets of files:

- **Application, mapping, and architecture models:** ETH defined a set of three different meta-models so as to describe KPN applications, the targeted architecture, and the desired mapping of the application onto the architecture. The application model is supposed to be written by the application designer, the architecture model by the hardware designer, and the application mapping by the system designer.
- **Application's task sources:** ETH also defined a set of API in order to provide KPN-compatible services, such as *DOL_create*, *DOL_read*, or *DOL_write*. These methods are made available to application programmers in both C and C++ programming language.

The main idea behind DOL is to allow application programmers to execute their application on a high-level simulation model of the target hardware, apply performance optimization algorithms, and modify their mapping. Once the application's mapping has been adjusted, our design flow is used to produce the binaries for the target platform.

5.2.4 Operating strategy

The D940 chip has been designed so as to be operated using either a standalone approach or a GPOS-based approach, as detailed in sections 2.1 and 2.2: ATMEL distributes a port of a GNU/Linux distribution for the D940 ARM processor, a device driver for the DSP, and a BSP compatible with both the ARM and the DSP. Hardware paths available for software communications are also a good indicator of a software design approach selected for a particular hardware.

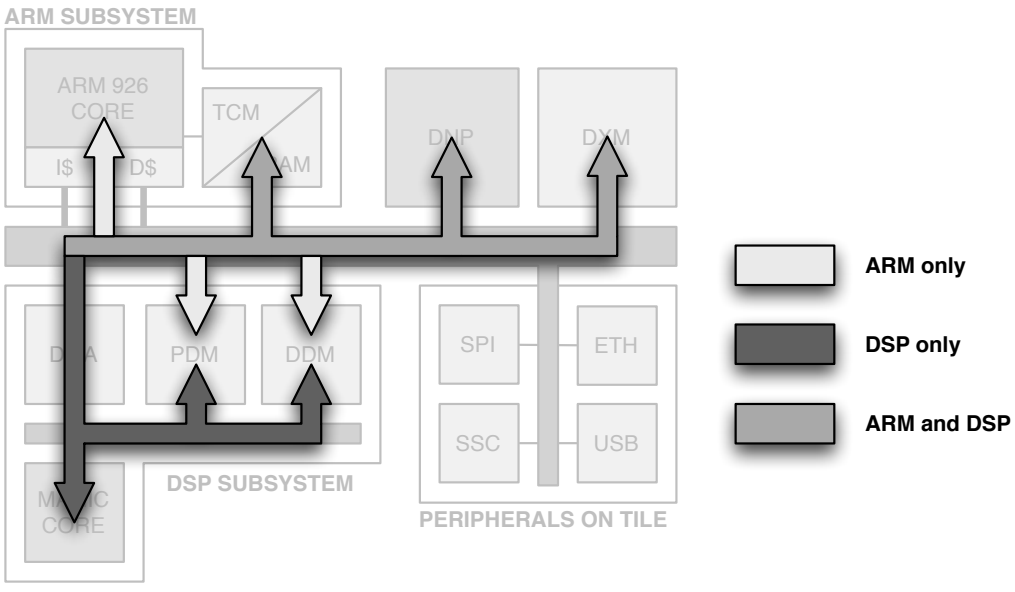


Figure 5.4: D940 hardware paths for software communications.

We define hardware paths as different combinations of memory buffers and data transfer elements involved in a data transfer during software communications. Data transfer elements are hardware devices able to transfer data, which includes DMA devices, communication devices such as the DNP, and processors. Figure 5.4 shows the different hardware paths available on the ATMEL D940. Light gray paths are reserved to the ARM processor, dark gray paths are reserved to the DSP processor, and medium gray paths can be used by both processors.

Two points are particularly noticeable on this figure: 1) the DSP can only implicitly access its own local memories, any other accesses must be explicitly done using its DMA controller; 2) the ARM processor has a complete control over the DSP. In this configuration, one hardware path becomes obvious: `ARM:DXM:DMA:DDM:MAGIC`, which corresponds to the type of communication that usually takes place between a GPOS and a co-processor.

Unfortunately, the GPOS approach is not fit to support a distributed programming model such as DOL. Indeed, **no distinctions in terms of communication or execution are made** between the DOL processes, should they be mapped on the ARM processor or the DSP. This particularly means that a DOL process mapped on the DSP **should share the same freedom of expression** as a process mapped on the ARM, which is not possible with a GPOS-based approach since processes mapped on the DSP could not, for instance, initiate communications or directly access the peripherals.

The approach presented in this thesis is able to abstract the details from the computation and the communications contained in the input model and efficient enough to operate two radically different architectures such as a RISC processor and a DSP. As a result, we considered the D940 as a fully distributed architecture and used a network of component-based software stacks to provide the same programming environment for both processors.

HAL components and compilation tool-chains

With the concept of HAL, the component-based software stack offers a solution to the abstraction of a process' execution (cf chapter 4). At that time, our framework already contained a HAL component for the ARM926EJ-S processor. We merely adapted it to the D940 platform. In particular, we modified the interrupt management functions to handle the D940's advanced interrupt controller and we enabled the processor's caches. On the DSP side, we had to create a HAL component from scratch. Due to the architectural limitations of the signal processor, the exercise was different from what we experienced with RISC processors:

- CPU I/O functions had to take into account the incapacity of the processor to directly address the whole memory map. Hence, each I/O operation had to be translated into a DMA request.
- The interrupts subsystem uses registers to store the service routines. Fortunately, the DSP compiler provides architecture-specific routines to access these registers. This was also true for the synchronization mechanism since the DSP embeds a register of 16 hardware semaphores.
- 128-bit VLIW assembly language with 6 to 8 instruction slots can be pretty unhinging for the rookie and is certainly not for the faint at heart. Nevertheless, we had to implement a few functions using the DSP's assembly language, such as its starts function. Only to please the connoisseur, some assembly code from the operating system is given in example 5.1.

Example 5.1 shows that control-oriented code does not take advantage of VLIW architectures since, on average, only 50% of the execution slots are used simultaneously. As a consequence, small and efficient operating systems are required so as to limit the impact of control-oriented code on the overall processor performance.

Example 5.1 Peek at the ATMEL D940 DSP assembly code.

```
 1      -      :      -              -      : DATA[1.A] = RFR2 = IADD(RFL64,RFR0) : - - - -
 2      -      :      -              -      :          -  COND = ILE(RFL66,RFL64) : - - - -
 3      -      :      -              -      : DATA[2.A] = RFR2 = IADD(RFL64,RFR0) : - - - -
 4  RFL65=0xff :      -              -      : DATA[0.A] = RFR2 = IADD(RFL5,RFR0)  : - - - -
 5      -      :      -              -      : DATA[677] = RFR2 = IADD(RFL64,RFR0) : - - - -
 6  RFL8=0x9   :      -              -      : TMP1=0.A RFL9 = IADD(RFL5,RFL0)     : - - - -
 7      -      :      -              -      : DATA[1.A] = RFR2 = IADD(RFL5,RFR0)  : - - - -
 8      -      : 0.A = DATA[15.A]    -      :          -              -          : - - - -
 9      -      : RFL65 = DATA[1.A]   -      :          -              -          : - - - -
10  4.A=1.A    :           -         -      :          -              -          : - - - -
11      -      : rf_inc = DATA[4.A]  -      :          -              -          : - - - -
12      -      : DATA[15.A+0] = 1.A  -      :          -              -          : - - - -
```

The tool-chain used to compile the application's component graph on the ARM was a plain GNU-based tool-chain composed of the *binutils* and the *GCC* compiler, to which we combined the RedHat Newlib. The tool-chain used to compile the application's component graph on the DSP was a proprietary tool-chain provided by Target Technology. Although both of them are quite dissimilar, they were both integrated to our design flow.

Nonetheless, there is one exception concerning the linker command file (section 3.1.7). While GNU LD offers its robust *ldscript* language, *bridge*, the Target's linker, offers a meager configuration file that provides our solution with only half the features it requires, hence forcing us to improvise. We overcame *bridge*'s restrictions by coupling its configuration file with a simple C file containing variable definitions. We also integrated our own C99 library into Target's tool suite, since the one supplied with the compiler was compatible with a standalone development approach only. The compilation of the other components, even the operating system, was successful with both compilers, since both understand C99 and APES's components are 100% C99 compliant.

Extension of the operating system

Once we found a solution to abstract the hardware dependences from the process' execution, we had to find a way to abstract the communications between processes. In the DOL files, communications are described using hardware paths (figure 5.4), distinguished in two categories: the intra-processor paths (e.g. `ARM:SRAM:ARM` and `MAGIC:DDM:MAGIC`) and the inter-processor paths (e.g. `ARM:DXM:DMA:DDM:MAGIC`). While intra-processor paths only require some basic synchronization such as semaphores, using inter-processor paths is more tricky since it usually involves interrupts, inter-processor synchronization, and data type translation. In addition, several protocols could be used over these channels and implementing every combination would not have solved the abstraction issue.

The approach we decided to adopt was to offer a standard *open*/*read*/*write* approach and encapsulate the couple $(protocol, path)$ into a device driver. At that time, DNA-OS was only composed of the core component and the memory manager component. Consequently, we

specially developed the Virtual File System (VFS) component, the root and device file systems, and the device driver interface. We implemented the *rendez-vous* and FIFO communication protocols on both the intra-processor and the inter-processor paths as two device driver components.

Finally, we integrated these components into our software framework. We implemented the functions of the DOL API accordingly so as to take advantage of these new components: functions such as *DOLcreate_port*, *DOL_read*, and *DOL_write*, which are directly interfaced with the *open*/*read*/*write* system calls to access the device drivers. Therefore, the same DOL process can be compiled and executed on both processors indifferently and access the same communication channels.

5.2.5 Execution of our design flow

For this experiment, the INFN first mapped the basic LQCD unit (depicted in figure 5.2) on a single D940 chip and validated the adaptation with a small data set (figure 5.5). Then, they extended the mapping to the ShapOtto platform, multiplying both the basic unit and the data sets. The basic LQCD unit actually contains two separate applications. The first application gathers the DISPATCH and the GATHER tasks, and runs on the ARM processor. The second application contains only the QPHI task and runs on the MagicV DSP. Both applications rely on the DOL to perform intertask communications. In turn, DOL uses a zero-buffer *rendez-vous* driver, both for local and global communications.

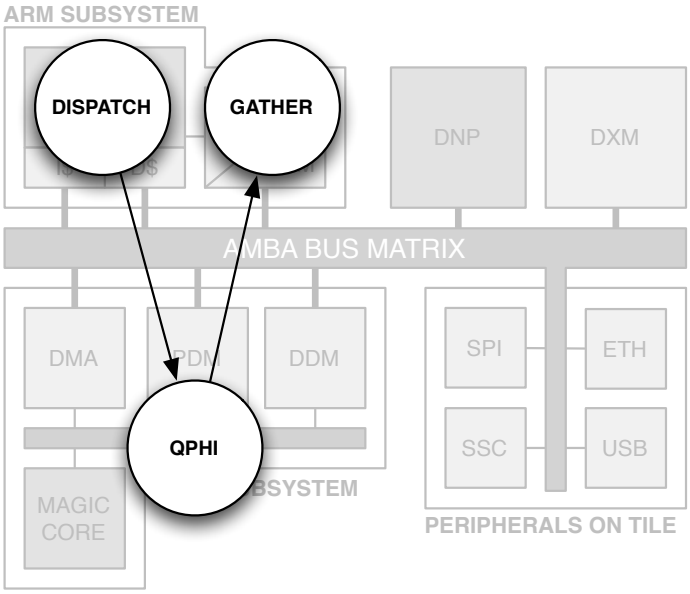


Figure 5.5: Mapping of the LQCD adaptation on a single D940.

Although the software architecture used in APES allowed the developers of the LQCD application not to deal with details related to the underlying processor architecture, they still had to take into consideration the limited amount of memory available on the DSP. Working with limited amounts of hardware resources still requires special attention from the developer. In addition, the development of the device drivers was quite straightforward since 95% of the code is identical on both ARM and DSP versions.

INFN was in charge of writing the XML files for their application. Since the DOL's application model only addresses tasks layout and communication structures, they also had to provide the source files of their application's processes. Starting from these files, GECKO produces two directories — one for each processor — where it generates the application's initialization file (a digest of these files is given in examples 5.2 and 5.3), the tool-chain's *Makefile*, and copies the provided task source files. Since the component selection was not yet automated, the application developers also had to write the component configuration file as well as the linker script. At that point, a simple call to the *make* command in each of these directories produces the application's binaries.

Example 5.2 LQCD's initialization file for the ARM processor.

```
#include <dol_com.h>
#include <dol_task.h>
...

/* DOL processes declaration */

DOLProcess intertile_mixer_0 = ...;
DOLProcess memory_dispatcher_0 = ...;
DOLProcess output_gatherer_0 = ...;
DOLProcess tile_initializer = ...;
DOLProcess tile_reassembler = ...;
...
int main (void)
{
 /* DOL port creations */
 DOLcreate_port (& intertile_mixer_0, 7, 7);
 DOLcreate_port (& memory_dispatcher_0, 2, 2);
...

 /* DOL port instantiations */

 DOLinit_port (& intertile_mixer_0, INPORT, 0, "/devices/rdma.7", 2, 0,6);
 DOLinit_port (& intertile_mixer_0, INPORT, 0, "/devices/rdma.9", 2, 1,6);
 DOLinit_port (& memory_dispatcher_0, INPORT, 0, "/devices/rdv.3", 0);
 DOLinit_port (& memory_dispatcher_0, INPORT, 1, "/devices/rdv.0", 0);
 DOLinit_port (& output_gatherer_0, INPORT, 0, "/devices/d940_rdv.0", 0);
...

 /* DOL tasks creations */

 DOL_create (& intertile_mixer_0, "intertile_mixer_0", 1, 0);
 DOL_create (& memory_dispatcher_0, "memory_dispatcher_0", 1, 0);
 DOL_create (& output_gatherer_0, "output_gatherer_0", 1, 0);
 DOL_create (& tile_initializer, "tile_initializer", 0);
 DOL_create (& tile_reassembler, "tile_reassembler", 0);
 DOL_join (& tile_reassembler);
 return 0;
}
```

Examples 5.4 and 5.5 present the configuration files' contents for both the ARM processor and the DSP. The first five lines of these files set the components to be used for the processor abstraction, the operating system, the task library, the communication library, and the platform abstraction. The last two lines declare which drivers have to be included in the final binaries. It is important to note that, besides the processor abstraction and the operating system component the elements used in both configurations are identical.

Example 5.3 LQCD's initialization file for the DSP.

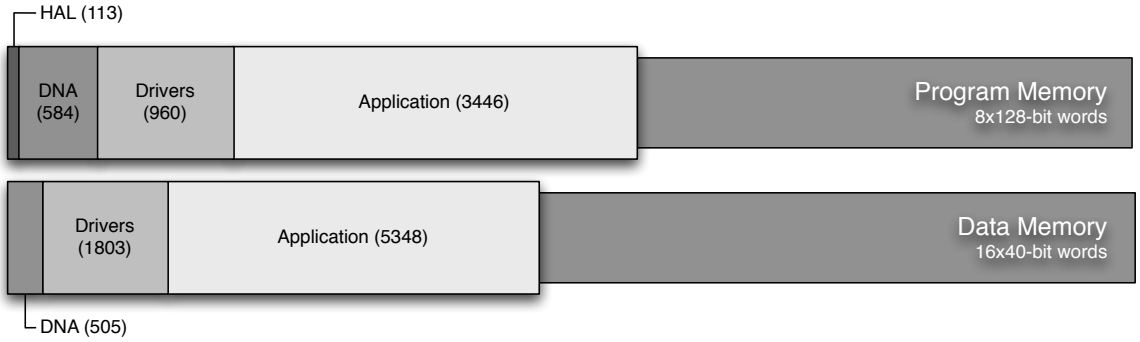
```
#include <stdlib.h>
#include <string.h>
#include <dol_com.h>
#include <dol_task.h>
#include <string.h>

/* DOL processes creation */

Qphi_compute_State Qphi_compute_0_state;
DOLProcess Qphi_compute_0 = ...;

int main (void)
{
  /* DOL port creations */

  DOLcreate_port (& Qphi_compute_0, 1, 1);

  /* DOL port instantiations */

  DOLinit_port (& Qphi_compute_0, INPORT, 0, "/devices/d940_rdv.1", 0);
  DOLinit_port (& Qphi_compute_0, OUTPORT, 1, "/devices/d940_rdv.0", 0);

  /* DOL static schedule */

  Qphi_compute_0 . wptr . indexes . x = 0;
  Qphi_compute_0 . wptr . is_scheduling_static = 1;
  Qphi_compute_0 . init (& Qphi_compute_0);

  while (1)
  {
    if (! Qphi_compute_0 . wptr . cancel_state)
    {
      Qphi_compute_0 . fire (& Qphi_compute_0);
    }
  }

  return 0;
}
```

Example 5.4 Configuration file for the ARM processor.

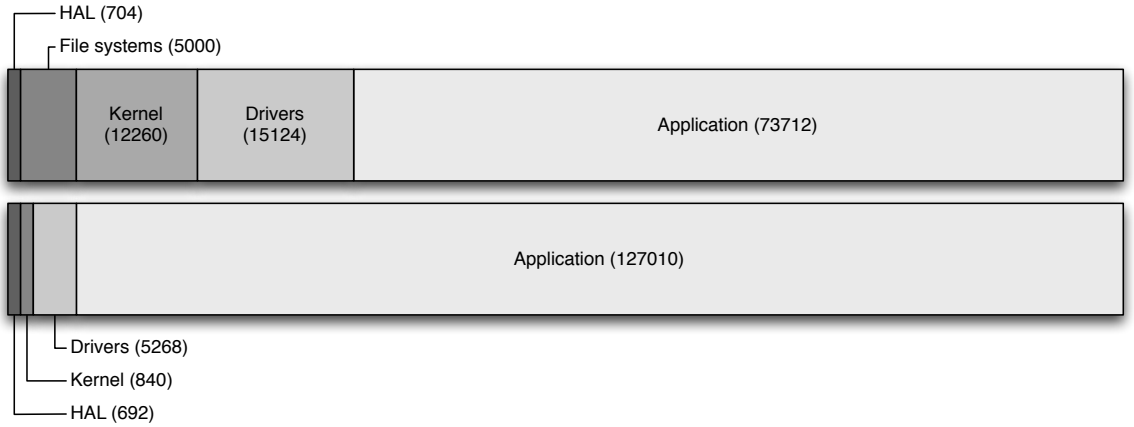
```
export TARGET_CAL="ARM_9_D940"
export TARGET_SYSTEM_KSP_OS="SYSTEM_KSP_OS_DNA"
export TARGET_SYSTEM_KSP_TASK="SYSTEM_KSP_TASK_DOL"
export TARGET_SYSTEM_ASP_COM="SYSTEM_ASP_COM_DOL"
export TARGET_SYSTEM_SSP_PAL="SYSTEM_SSP_PAL_ATMD940"
export DNA_CHANNEL_DEVICES="d940_rdma d940_rendezvous rendezvous"
```

Example 5.5 Configuration file for the DSP.

```
export TARGET_CAL="ATMEL_MAGICV"
export TARGET_SYSTEM_KSP_OS="SYSTEM_KSP_OS_DNA_MONITOR"
export TARGET_SYSTEM_KSP_TASK="SYSTEM_KSP_TASK_DOL"
export TARGET_SYSTEM_ASP_COM="SYSTEM_ASP_COM_DOL"
export TARGET_SYSTEM_SSP_PAL="SYSTEM_SSP_PAL_ATMD940"
export DNA_CHANNEL_DEVICES="d940_rendezvous"
```



(a) DSP binary



(b) ARM binary: *text* section (upper graph) and *data* section (lower graph), in bytes.

Figure 5.6: Binary layouts of the LQCD application.

On the ARM processor, the application required the following components: the full-featured Core, the full-featured VFS, and the Memory Manager from the DNA-OS, as well as the ARM HAL, the default file systems, the D940 platform, the D940 system driver, the *rendez-vous* driver, and the DOL library. On the DSP processor, the application required the following components: the single-threaded Core, the namespace-based VFS, and the Memory Manager from the DNA's Not just Another Operating System (DNA-OS), as well as the MagicV HAL, the default file systems, the D940 platform, the D940 system driver, the *rendez-vous* driver, and the DOL library. There are only three differences between these two configurations:

- **The Core component:** on the ARM processor it is simple to implement a complete thread management with a scheduler and context switches, whereas it is slightly more complicated and probably inefficient to do the same on a DSP with a register file of 256 units. Therefore, we implemented a Core component that supports only one execution thread. If more than one DOL process is mapped on the DSP, their execution is statically scheduled by the code generator even though the presence of deadlocks induced by communication-based data dependencies is not verified.
- **The VFS component:** due to the memory limitations of the DSP we could not directly use the full implementation of the VFS. As a consequence, we developed a reduced implementation of its API based on a namespace principle.
- **Hardware Abstraction Layer:** each software stack uses the HAL component suited to its target processor architecture.

Since the underlaying concept of our approach remains the same for both platforms, the application's developers were able to scale the one-tile configuration to a ShapOtto-compatible configuration very rapidly. This adaptation has been made by the INFN in collaboration with Alexandre Chagoya-Garzon as a part of his research topic concerning the software generation and inter-tile communications on multiple-tile systems. He extended our approach to support the code generation on the ShapOtto platform and wrote a new communication channel driver that makes use of the DNP and proposes two protocols: RDMA and Eager [17].

Besides the inclusion of Alexandre's communication driver, nothing had to be changed in the per-tile configurations. Only the process network and the mapping files had to be modified. The process network features two more processes per tile: the Intertile Mixer and the Tile Reassembler. These two processes implement a data synchronization protocol to ensure the validity of the computed sub-lattices with their direct neighboring tiles. The mapping files includes new DNP-based communication channels that connect the Intertile Mixer from one tile to the Tile Reassembler from another tile.

Figure 5.6 presents the memory footprint for both ARM-side and DSP-side binaries. On both graphs, the most obvious data is the very small amount of memory required by our framework. For each instance, the required memory does not exceed 30% of the overall memory footprint. Performance-wise, we unfortunately did not produce any execution result to present. This is mainly due to the initial challenge of the SHAPES project, which was the automation of the application's development for complex, multiprocessor, heterogeneous platforms. However, several synthetic evaluations showed that the performance of an application running on both processors of a single tile is relatively close to ideal in terms of both computation and communication.

5.3 Second experiment

In this experiment, we used our framework to adapt two complex, parallel applications, an MJPEG video decoder and an edge filter based on the Sobel operator, on a heterogeneous, multiprocessor, simulation platform. We first present the applications, then we present the hardware platform, and finally we discuss the adaptation and give several results.

5.3.1 Motion-JPEG decoder

The Motion-JPEG (MJPEG) video format is composed of a succession of JPEG still pictures. It is used by several digital cameras to store video clips of a relatively small size. With MJPEG, each frame of video is captured separately and compressed using the JPEG algorithm. JPEG is a lossy compression algorithm, meaning that the decompressed image is not totally identical to the original image. Its goal is to reduce the size of natural color images as much as possible without affecting the quality of the image as experienced by the human eyes.

The JPEG compression algorithm splits an image in blocks of 8×8 pixels, then translates each block into the frequency domain, eliminates the high frequencies using a per-image filter, and compress the resulting blocks using a Huffman encoding with a per-image dictionary. The blocks are also called macro-block or Minimum Coded Unit (MCU). The resulting bit stream is made of sequences of raw data separated by markers that identify data. Inversely, the JPEG decompression algorithm is composed of five consecutive operations: Variable Length Decoder (VLD), Inverse Zig-Zag Scan (unZZ), Inverse Quantization (IQ), Inverse Discrete Cosine Transform (IDCT), and Block Reordering (BR). The VLD operation uses the Huffman

algorithm to extract the 8×8 MCU. The unZZ operation reorders the data of a MCU following a zigzag order. The IQ operation applies the quantification factors contained in the Joint Photographic Experts Group (JPEG) header to each element of a MCU. The IDCT operation translates each element of a MCU from the frequency domain to the color domain. The BR operation reconstructs the final picture.

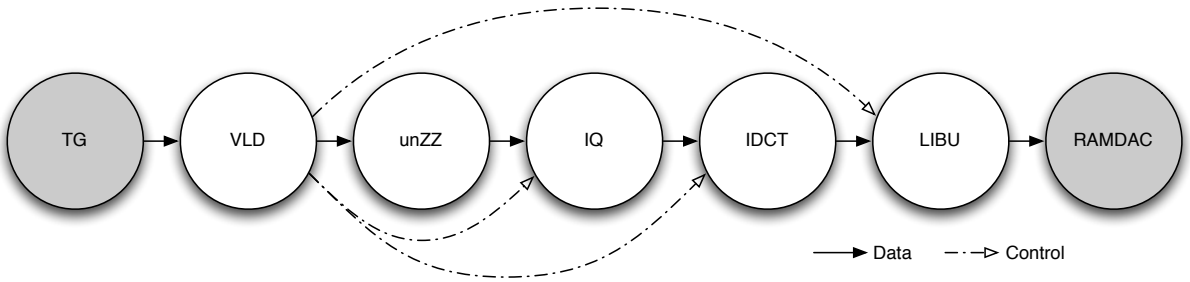


Figure 5.7: Original MJPEG algorithm.

The version of the MJPEG we used in our experimentation is a modified version of the pipelined version of the algorithm that has first been developed at the Paris VI University. The original version features seven parallel tasks, five of which corresponding to a basic computation block of the algorithm (figure 5.7). The Traffic Generator (TG) task is responsible for feeding the pipeline with raw data from a MJPEG movie. The VLD task is reponsible for decompressing the raw MJPEG flow according to the Huffman's algorithm. It then dispatches the data to the IQ task or the unZZ task accordingly. The unZZ task is responsible for applying an inverse zig-zag scan to the current data received from the VLD task. The IQ task is responsible for applying the adequate quantization table to the data received from the unZZ task. The IDCT task is responsible for decoding the data received from the unZZ task from the frequency domain to the color domain. The Line Builder (LIBU) task is responsible for reassembling the data received from the IDCT task into a displayable picture. The RAMDAC task is responsible for displaying the picture received from the LIBU task.

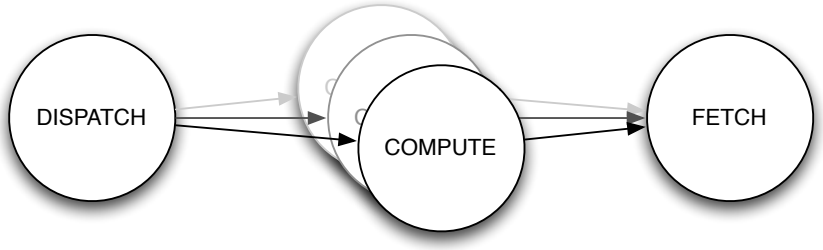


Figure 5.8: Modified MJPEG algorithm.

This version of the decoder is not satisfactory: its performance is limited due to the disproportioned number of tasks compared to their average workload, it only supports one kind of sub-sampling, gray-scale pictures, and it does not allow for the parallelization of the IDCT. As a consequence, we (Patrice Gerin and myself) modified the original algorithm in order to include these features. We added the support of color pictures and sub-sampled compressions. We modified the length of the data packet exchanged from one macro-block to one line of macro-blocks. We merged the TG, VLD, unZZ, and IQ tasks into a single task named DISPATCH. We also merged the LIBU and RAMDAC tasks into a single task named FETCH. This

design (depicted in figure 5.8) allows us to parallelize the processing of the IDCT in multiple tasks depending on the number of available processors.

5.3.2 Sobel operator

The Sobel operator is used in image processing, particularly within edge detection algorithms. Technically, it is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel operator is either the corresponding gradient vector or the norm of this vector. The Sobel operator is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical directions and is therefore relatively inexpensive in terms of computations. On the other hand, the gradient approximation which it produces is relatively crude, in particular for high frequency variations in the image.

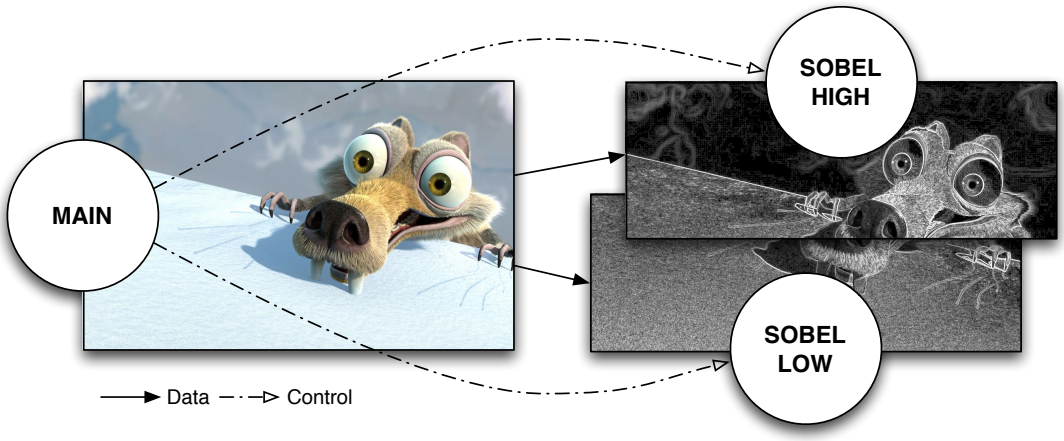


Figure 5.9: Implementation of the Sobel operator.

In simple terms, the operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result therefore shows how "abruptly" or "smoothly" the image changes at that point, and therefore how likely it is that that part of the image represents an edge, as well as how that edge is likely to be oriented. In practice, the magnitude (likelihood of an edge) calculation is more reliable and easier to interpret than the direction calculation.[1]

We implemented this algorithm in an event-triggered fashion. It contains three tasks. The MAIN task gathers a decoded image and prepares the Sobel buffers depending on the picture format. It then awakes the SOBEL LOW and SOBEL HIGH tasks that apply the Sobel operator on the lower part and the higher part of the picture respectively. These tasks are suspended upon completion, and the MAIN task merges and displays the results (figure 5.9).

5.3.3 HeSyA platform

In order to stress each aspect of our contribution, we needed a complex, heterogeneous, hardware platform on which we could simultaneously run both the MJPEG application

[1] Source: [2], page 135, section 7.3.2

and the Sobel filter. For that purpose, we built the Heterogeneous Symmetric Architecture (HeSyA) platform (figure 5.10) using the SoCLib SystemC library [54].

HeSyA contains four distinct and independent subsystems. The MIPSR3000 subsystem contains three MIPS R3000 and a local memory connected to a crossbar. This subsystem has its own address space and runs as a local SMP cluster. The ARM11 subsystem contains two ARM 11 core and a local memory connected to a crossbar. In the same fashion as the MIPS subsystem, it has its own address space and runs as a local SMP cluster.

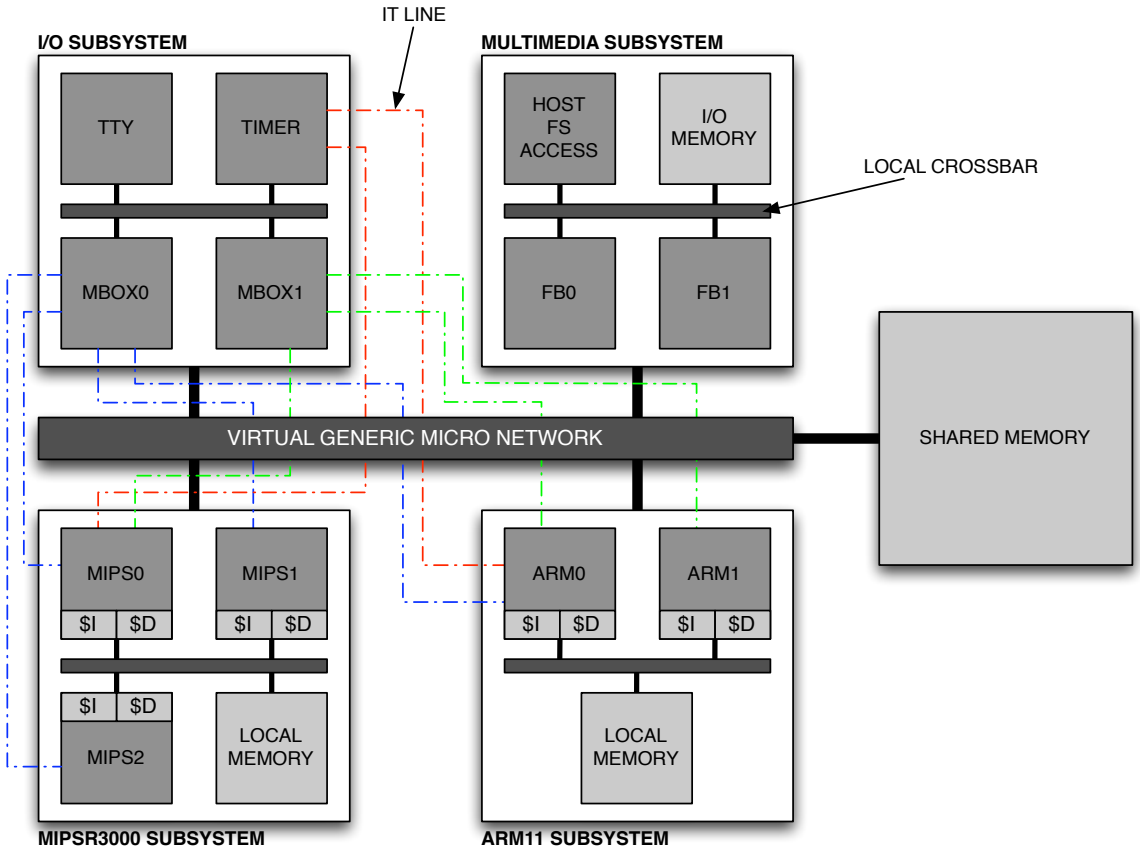


Figure 5.10: HeSyA platform.

The I/O subsystem contains peripherals required to perform standard I/O operations: a terminal sink (TTY), a timer (TIMER), a mailbox engine dedicated to the MIPS subsystem (MBOX0), and a mailbox engine dedicated to the ARM subsystem (MBOX1). The mailboxes are configured as follows: for each processor subsystem, the n first entries correspond to the n local processors, while the $n+1$ entry corresponds to the processor with identifier 0 of the peer subsystem.

The multimedia subsystem contains peripherals required by the multimedia applications: a host file system access (HOST FS ACCESS) to read data from the MJPEG movie file and its associated I/O memory, a framebuffer dedicated to the MIPS subsystem (FB0), and a framebuffer dedicated to the ARM subsystem (FB1).

As stated before, each subsystem is independent. This means that, by default, none of their resources are shared with the other subsystems. To circumvent this feature, some address segments are shared: the I/O subsystem shares the TTY and the TIMER with both processor subsystems, the MBOX0 is shared only with the MIPS subsystem, and the MBOX1 is shared

only with the ARM subsystem; the multimedia subsystem shares the HOST FS ACCESS and the FB0 with the MIPS subsystem, and FB1 is shared with the ARM subsystem. In addition, we included a shared global memory that can be used to transfer data between the two processor subsystems.

5.3.4 Programming model

Besides restructuring the thread organization and improving its features, we did not change the base source code of the MJPEG application: the original version uses the PThreads API, so does the one we modified. We implemented a PThreads library on top of the DNA-OS thread API in order to integrate this application in our framework. The original MJPEG also uses a specific communication library (developed at Paris VI) called KPN communication channels. This library implements hardware- and software-based FIFO communication protocols and provides a specific API to interact with them. Since it directly manipulates hardware devices, this KPN library is not compatible with our software framework. In order to integrate the MJPEG application without changing its source code, we implemented a communication library API-compatible with the original one but interacting with the DNA-OS VFS to perform communications.

As a result, applications using functions from the PThreads library, the standard C library, or the KPN library can be ported on any kind of platform supported by our software framework. Moreover, this kind of application can indifferently be recompiled and executed on both POSIX-compatible operating systems and embedded platforms using the DNA-OS. This is particularly the case for the SPLASH-2 benchmark suite [94] or the Tachyon ray tracer [85], both used with our software framework in Pierre Guironnet-de-Massas' and Patrice Gerin's works on simulated MP-SoC using multiple instances of ARM, MIPS, and SPARC processors [28, 32].

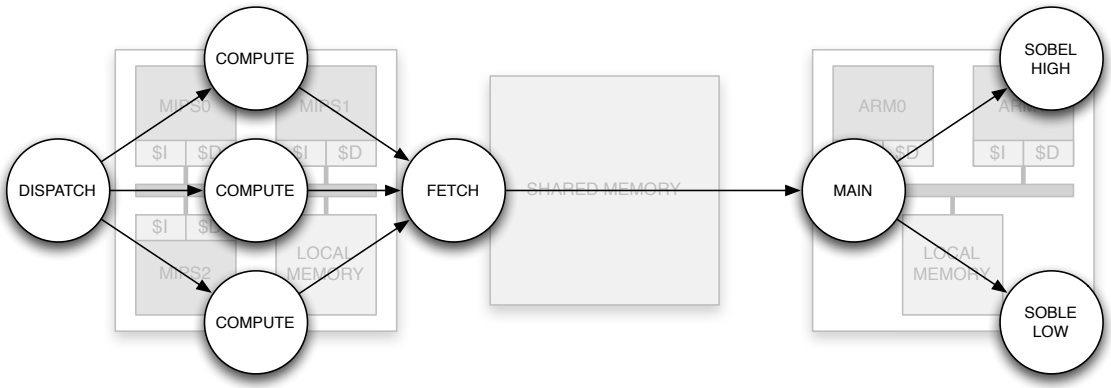


Figure 5.11: Mapping of the MJPEG and the Sobel operator on the HeSyA platform.

5.3.5 Execution of our design flow

Firstly, we mapped the MJPEG application on the MIPS subsystem and the Sobel filter on the ARM subsystem (figure 5.11. Each local communication is performed using zero-buffer *rendez-vous* protocol. The local synchronizations are carried out by semaphores. The global communication is performed using a buffered *rendez-vous* protocol. The global synchronization is carried out by inter-processor interrupts. Each application has been encapsulated in

an OCM component. Secondly, we arbitrarily switched the processor subsystems between the two applications. To do so, we simply changed the restriction on the HAL components and updated the compilation information for each application.

The goal of this operation is to highlight the portability and automation enabled by our approach. We show, at the end of this section, a binary footprint comparison between the two configurations. Since the targeted applications were already shaped in component forms, we did not have to run the GECKO to generate their source code. We merely had to write their corresponding interface descriptor so as to integrate them with our framework. Starting from this point, the production if the binary code is purely automated. The following subsections present the component graphs for each application.

Construction of the MJPEG decoder application

The MJPEG decoder needs the MIPS processor and the SoCLib platform support. It depends on the PThreads library to create its threads. Each thread can migrate freely between each processor. The operating system has been configured with a FIFO scheduler, as well as drivers for the HOST FS ACCESS (HPD), the local *rendez-vous* communication channel (RDV), the framebuffer (SPD), and the global *rendez-vous* channel (HCD).

This application also relies on a "message passing"-like communication library (KPN) to perform its inter-thread communications. A full row of macro-block is exchanged between each task. Due to the Huffman decoder, the movie file is accessed byte-per-byte. Therefore, we implemented an internal buffer manager in order to speed-up single-byte accesses and reduce the system call overhead. The Newlib C library provides the common C functions.

Figure 5.12 shows the component graph of the MJPEG application. The vertices of the graph represent the components and the arcs represent the dependencies between components. This graph has been built automatically starting from the application's component. It is easy to see the separation between each layer of our software framework (user in light gray, operating system in medium gray, and HAL in dark gray). Thanks to this separation, porting this application on a different processor or on a different platform is just a matter of switching components since no other dependencies than the functional dependency is involved in the component graph construction. We actually ported this application on the ARM subsystem

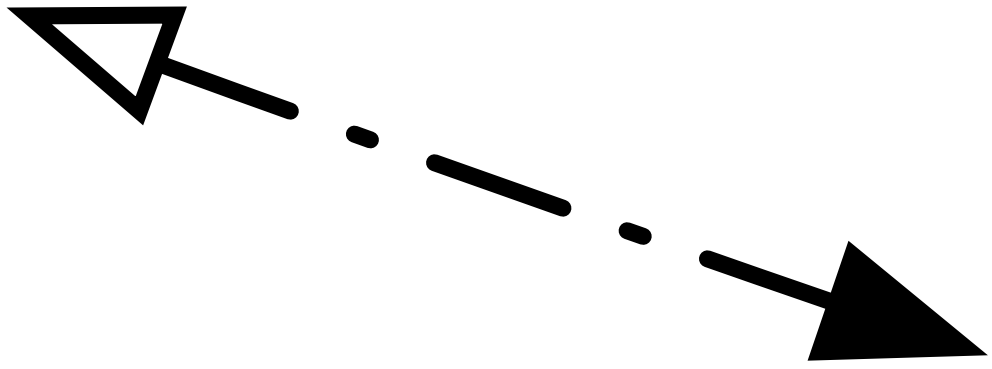
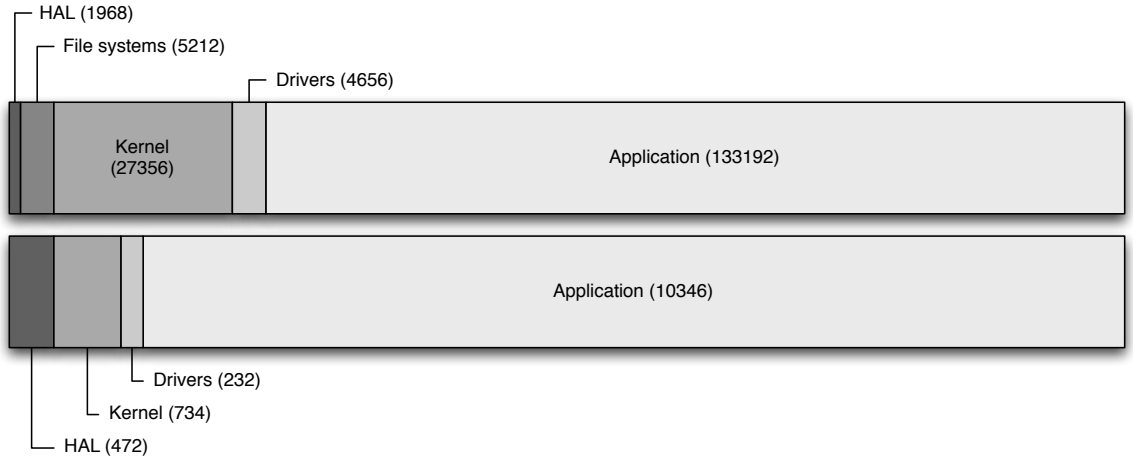
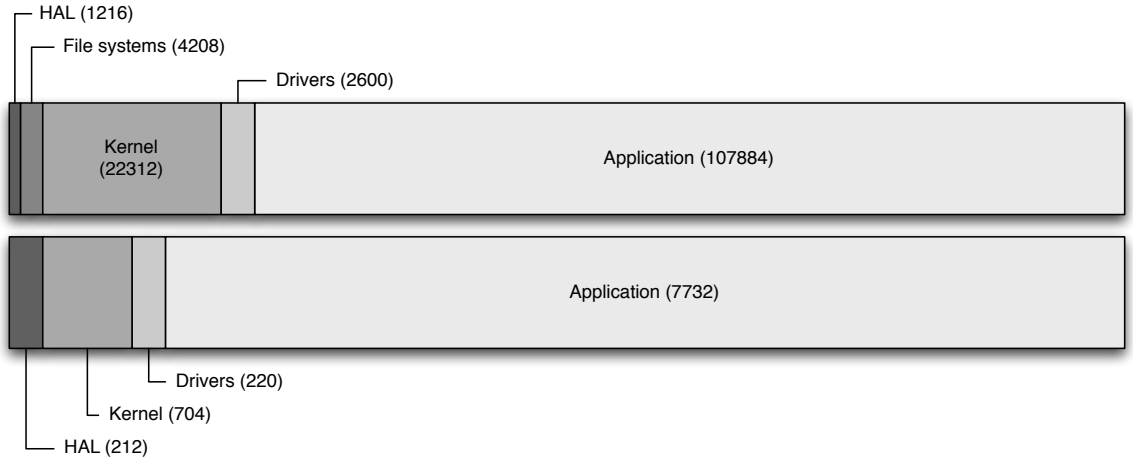


Figure 5.12: Component graph of the MJPEG decoder.

of the HeSyA platform: the entire operation took approximatively three minutes between the reconstruction of the component graph and the compilation of the components.



(a) MIPS binary: *text* section (upper graph) and *data* section (lower graph), in bytes.



(b) ARM binary: *text* section (upper graph) and *data* section (lower graph), in bytes.

Figure 5.13: Binary layouts of the MJPEG application for both MIPS and ARM subsystems.

Figures 5.13(a) and 5.13(b) present the memory footprints of the MJPEG application on both MIPS and ARM processor subsystems. Despite small variations due to the different instruction set architectures and the different number of target processors, the MIPS and ARM layouts are quite similar. The size of the software framework's overhead is noteworthy: in both cases, it does not exceed 25% of the whole binary size.

Construction of the Sobel operator application

The Sobel operator requires the ARM processor and the SoCLib platform support. It depends on the native thread library to create its threads. Each thread is dedicated to a specific processor: SOBEL HIGH runs on processor ARM0 and SOBEL LOW runs on processor ARM1. The MAIN thread can migrate freely between the processors. The operating system has been configured with a FIFO scheduler, as well as drivers for the local *rendez-vous* communication channel (RDV), the framebuffer (SPD), and the global *rendez-vous* channel (HCD, HPD). This

application also relies on a "message passing"-like communication library (KPN) to perform its inter-thread communications. The Newlib C library provides the common C functions.

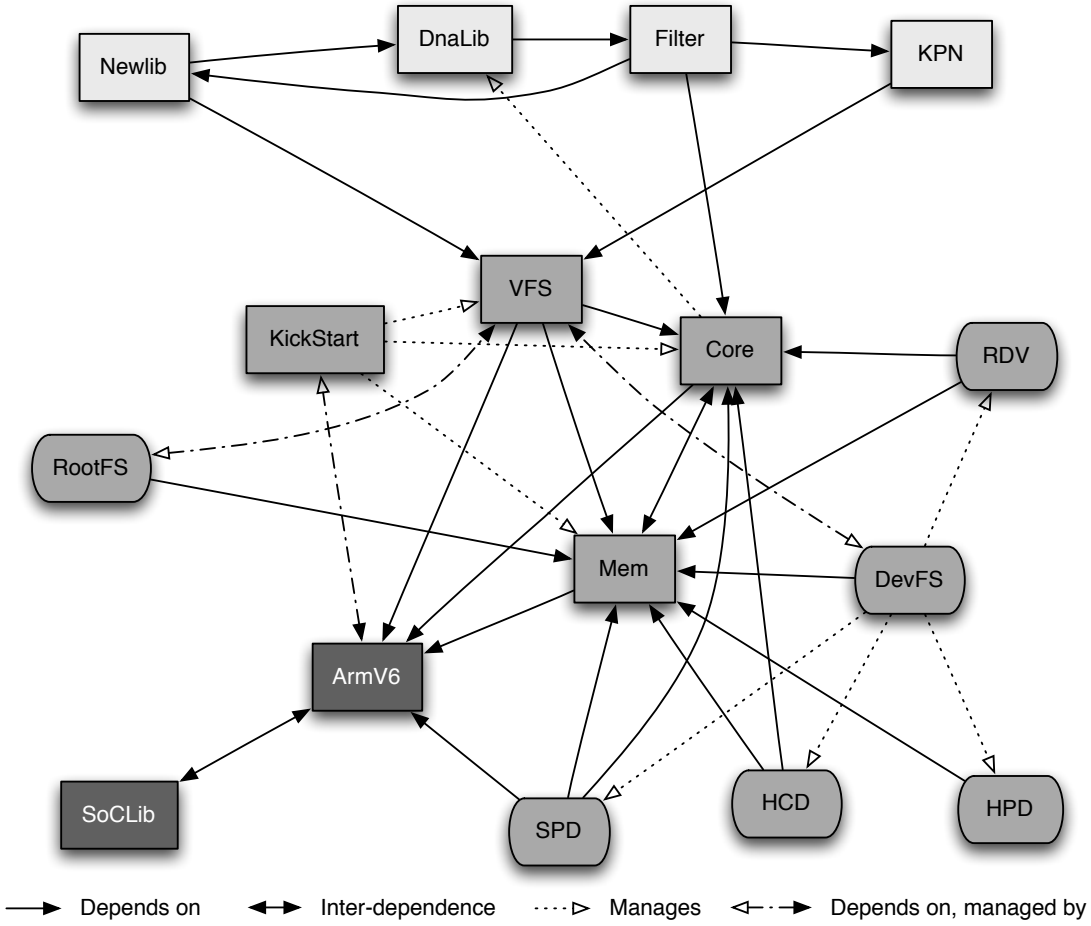


Figure 5.14: Component graph of the Sobel filter.

Figure 5.14 shows the component graph of the Sobel application. This graph has been built automatically starting from the application's component. As before, it is easy to see the separation between each layer of our software framework (user in light gray, operating system in medium gray, and HAL in dark gray). Like with the MJPEG application, porting this application on an different processor is only a matter of component switching. We ported the filter on the MIPS subsystem of the HeSyA platform and, as with the MJPEG application, the operation did not exceed a few minutes. The comparison of the binary layouts is not repeated here since the observations that can be carried out from such an experience are identical to the one made with the MJPEG application.

5.4 Current status of the framework

The purpose of our work was to provide an efficient embedded software design environment to embedded appliance designers. The experiments previously detailed, while demonstrating the qualities of our approach in terms of computer-aided design, failed at giving a clear status of our software framework as well as showing its raw performance. This section tries

to fill the gap and gives, for each major component of our framework, its current status and, if relevant, its associated performance.

5.4.1 Hardware abstraction layers

APES contains components for seven classes of processors: ARM v5 and v6, MIPS R4000, SPARC v8, Atmel MagicV DSP, Xilinx Microblaze, and Altera Nios II. While ARM and MIPS implementation are up-to-date, the other components need some revamp so as to be fully compatible with the latest processor interface. Concerning the platforms, APES currently contains three platform components: the Xilinx XUP, the Atmel D940 library, and the SoCLib simulation library. The D940 and the SoCLib components are rather complete since they implement all the functions of the platform interface and include definitions for most of their supported hardware. The XUP component is only in its minimal form. In the months to come, efforts will be put into the integration of the Texas Instruments (TI) BeagleBoard. HAL components will be written for the ARM v7 processor, the TI C64x DSP, and the OMAP3530 platform.

5.4.2 The DNA operating system

The Core component of the DNA-OS is complete. All the functions defined in its interface has been implemented, for a total of 2,115 lines of actual code and approximatively 13 KB depending on the target processor. It also fully supports multiprocessor configurations and can allocate a thread to a specific processor. This footprint correspond to the full component and can be easily reduced depending on the needs of the application by using some link-time garbage collection techniques. We developed several benchmarks so as to position our work relatively to existing embedded kernel implementations and we timed the following operations: thread switch (and not just the context switch), distributed thread switch (election of a thread allocated to a processor different from the one running `main`), interrupt latency (from the interruption to the Interrupt Service Routine (ISR)), and the minimum alarm period.

We executed these benchmarks on three different platforms: one MIPS-based simulation platform using the SoCLib library, one ARM-based simulation platform also using SoCLib, and the ARM processor of the ATMEL D940 platform. The execution characteristics of these platforms are:

- **MIPS-based SoCLib platform:** the timer configuration of the MIPS processor is defined as if the processor and the system bus were both running at 100 MHz. The memory access latency is set at 10 cycle/word and the processor features a 4-kiloword, write-through data cache.
- **ARM-based SoCLib platform:** the timer configuration of the ARM processor is defined as if the processor and the system bus were both running at 100 MHz. The memory access latency is set at 10 cycle/word and the processor features a 4-kiloword, write-through data cache.
- **ARM on the D940 SoC:** the ARM processor runs at 200 MHz and the SoC system bus runs at 100 MHz. The memory access latency is set at 11 cycle/word and the processor features a 4-kiloword, write-through data cache.

The benchmark results are presented in table 5.1. Unsurprisingly, the results on the SoClib platforms are rather similar. The difference (or the lack of difference) of thread switching performances between the ARM-based SoCLib platform and the D940 platform can be explained by the D940 ARM's higher memory access latency and by a certain inaccuracy level of the SoCLib's ARM Instruction Set Simulator (ISS). The result for the distributed thread switch is not available for the D940 platform since it features only one ARM processor.

	Thread switch	Distributed thread switch	Interrupt latency	Minimum alarm period
SoCLib ARM @ 100MHz	51 µs	53 µs	340 ns	25 µs
SOCLIB MIPs @ 100MHz	50 µs	54 µs	520 ns	30 µs
D940 ARM @ 200MHz	44 µs	*n/a*	180 ns	15 µs

Table 5.1: DNA-OS core component performance.

Table 5.2 presents results for existing operating-system solutions. Although most of the figures gathered in table 5.2 come directly from the developers of these respective works (and consequently scientifically questionable), they give a good idea of the relative position of our work compared to the others: the DNA-OS core component fares equivalently well compared to solutions such as QNX or VxWorks, renown for their good performances.

	QNX Pentium @ 200 MHz	**LynxOS** 68030 @ 25 MHz	**RTAI (Linux)** Pentium @ 100 MHz	**VRTX** 68030 @ 25 MHz	**VxWorks** 68030 @ 25 MHz
Context switch	2 µs	180 µs	4 µs	80 µs	110 µs
Interrupt latency	2,600 ns	13,000 ns	> 40,000 ns	4,000 ns	3,000 ns

Table 5.2: Other operating system performance.

The interrupt latencies of DNA-OS's Core component, while being closely linked the HAL implementation used, are rather short. Coupled with a *tickless* algorithm, the Core component provides a very sound and efficient alarm architecture. The only notable difference lays in the thread switch latencies. This is due to the fact that vendors usually measure *context switches* and not *thread switches per se*. However, we believed that giving the time of a thread switch would be more accurate, since a context switch is extremely processor dependent and does not reflect the actual operating system performance.

Additionally, we were able to measure, on a dual-processor SMP configuration of the SoCLib platforms, the time required to spawn a thread on a different processor than the one running `main`. On both processor architectures, it took only a few microseconds more. This overhead is due to the Inter-Processor Interrupt (IPI) framework, since the first processor has to send a `yield` message to the second processor using an interruption.

The VFS component is only at its early stage, although most of the basic functions are fully usable. Its complete architecture is divided in three subsystems:

- **The volume subsystem:** this subsystem deals with volumes, i.e. a couple formed of a block device and a file system. It is complete and fully functional.

- **The vnode subsystem:** this subsystem deals with virtual nodes, a generic metaphor of the *inode* notion. It is complete and functional.
- **The POSIX subsystem:** this subsystem contains the POSIX-compliant system calls. As of today, only the major functions have been implemented. Some work remains to be done for the VFS component to offer a complete POSIX interface.

In addition, the VFS has not yet been validated with multithread, parallel accesses nor with fully concurrent, multiprocessor resource sharing. This validation will be the next big development effort concerning this component.

The memory manager component is also in its early stages. It currently only implements and defines functions for the kernel allocator. The next development effort concerning this component will be to implement a generic page management layer compatible with both MMU and non-MMU processor subsystems.

5.4.3 Device driver and file system modules

In terms of device drivers, Application Elements for SoC (APES) contains support for most of the devices proposed in the SoCLib simulation library. APES also contains a block device driver for the multimedia card reader of the Xilinx XUP and several drivers for the Atmel D940's devices including the *ethernet* controller, developed by a Master student in less than a month. In terms of file systems, APES includes the obligatory `rootfs` and `devfs` modules as well as a FAT16 module. The development of modules for the FAT32 file system, the EXT2 file system, and the JFFS file system are currently in progress.

6
Conclusion and perspectives

This manuscript presented an efficient approach to rapidly develop embedded software for either homogeneous or heterogeneous Multi-Processor System-on-Chip. Based on the combination of a fully automated design flow and a well-furnished, component-based, software framework, this approach was designed to maximize the flexibility, portability, scalability, and automation criteria, fundamental in the domain of embedded software. We first started our dissertation with the following statement: modern embedded applications are complex and require a large amount of processing power that only MP-SoC architecture can provide. Then, we studied the most widespread embedded software development approaches. For each of them, we presented their related design flow and classified their advantages and drawbacks according to the four fundamental criteria. The primary conclusion of this analysis was that none of the three approaches were truly capable of producing efficient applications for MP-SoC architectures, although these approaches are the most popular.

The following chapter presented our first contribution: a novel embedded software design flow, built with the advantages of the common approaches in mind so as to score high marks in two of the four fundamental criteria: automation and flexibility. In the first section, we demonstrated how a first level of automation and flexibility can be addressed using an automatic code generator. In the second section, we first highlighted the need of a modular software approach to reach full flexibility and automation, and presented the two principal modular programming paradigm: Component-based Development (CBD) and Object-Oriented Programming (OOP). With none of these paradigms matching our needs, we presented Object-Component Model, an original mix between CBD and OCM that makes use of their theoretical compatibility to bring some OOP mechanisms to CBD. We showed how OCM can really tackle the flexibility and the automation of our design flow by using mechanisms such as automated component selection, component inheritance, and overriding polymorphism.

In chapter 4, we presented our second contribution: a component-based software framework using our OCM paradigm. We first showed how a clear separation between user-level, operating system, and Hardware Abstraction Layer components can score high marks in terms of portability and scalability. Then, we presented several components from our framework: the HAL components, the DNA-OS components, and some user-level components. Finally, we presented a set of different configurations that can be obtained with our framework as well as the tools used to build them. We concluded our dissertation with two major experiments,

both taking parts in either European or French national projects, that demonstrate the full potential of our contributions. The next points present the perspectives of our work.

Automation of the component generation

As of today, the automatic code generation of OCM components has not yet been implemented. Although the basis — the interface descriptor and the implementation descriptor — has been laid, some other meta-models are being designed to handle abstractions such as "programming language" or more simply "source file". Afterward, a model transformation technology — most probably the ATLAS Transformation Language (ATL), will be used to iteratively mold the initial descriptors into their final source code.

Toward a fully 2nd-generation μ-kernel architecture

The current architecture of the DNA-OS is only a few steps short of being a true 2nd-generation μ-kernel. The ultimate goal would be to take full advantage of the OCM in order to use the DNA-OS either as a part of the final application or as an independent μ-kernel. Thus, the same operating system could use all the capabilities of any kind of processor architecture and could perfectly fit all the needs of any kind of software application: one framework, one source code, an infinite number of efficient configurations.

Real-time scheduling and fault-tolerance algorithms

In the same spirit as the previous point, the core component of the DNA-OS needs to be extended in order to support real-time and fault-tolerance algorithms. The real-time feature is primordial in critical, time-constrained applications. These applications form an important subset of the embedded applications zoo, the design of which our approach is supposed to enhance. The fault-tolerant feature is becoming more and more fashionable especially in widely distributed appliances such as sensor networks and in large-scale configurations such as supercomputers which gather more than a thousand of processor core. Both of these features will be tackled during the upcoming European FP7 project EURETILE.

On- and off-chip generic task migration

This topic comes as an extension of the fault-tolerance feature proposed above. Ideally, faulty processors of a massively parallel architecture should be able to freeze the execution of their current processes, negotiate their migration with a compatible, neighboring processor, and put themselves offline. A generic approach to this problematic will be studied, and early implementations will be tested in the scope of the EURETILE project, where multiple instances (over a thousand) of the ShapOtto *tile* presented in chapter 5 will be interconnected in a large network of tori. In case of a hardware or a software fault, inter-processor and inter-tile task migrations are expected to take place. This applies for both the GPP and the DSP processor.

Annexes

Acronymes

AOP	Aspect-Oriented Programming
AIC	Advanced Interrupt Controller
ATL	ATLAS Transformation Language
AMBA	Advanced Micro-controller Bus Architecture
API	Application Programming Interface
APES	Application Elements for SoC
ARM	Advanced RISC Machine
AST	Abstract Tree Syntax
BSP	Board Support Package
BR	Block Reordering
CBD	Component-based Development
CDFG	Control Data Flow Graph
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSP	Communicating Sequential Processes
DAG	Direct Acyclic Graph
DDM	Data DSP Memory
DMA	Direct Memory Access
DNA-OS	DNA's Not just Another Operating System
DOL	Distributed Operation Layer
DNP	Distributed Network Processor
DSP	Digital Signal Processor
EEPROM	Electrically Erasable and Programmable Read-Only Memory
ETH	Eidgenössische Technische Hochschule
EXT2	Extended File System, 2^{nd}revision
FAT	File Allocation Table
FIFO	First In - First Out
FPGA	Field-Programmable Gate Array
FSM	Final State Machine
GCC	GNU Compiler Collection
GECKO	Générateur de Code
GFLOPS	Giga Floating Point Operations Per Seconds
GPP	General-Purpose Processor
GNU	GNU's not Unix
GPOS	General-Purpose OS
HA	Hardware Abstraction
HAL	Hardware Abstraction Layer
HeSyA	Heterogeneous Symmetric Architecture
HTG	Hierarchical Task Graph

HW	Hardware
I/O	Input/Output
IC	Integrated Circuit
IDE	Integrated Development Environment
IDL	Interface Definition Language
IFD	Interface Descriptor
IMD	Implementation Descriptor
IP	Intellectual Property
IQ	Inverse Quantization
IPI	Inter-Processor Interrupt
ISR	Interrupt Service Routine
ISS	Instruction Set Simulator
ITRS	International Technology Roadmap for Semiconductors
IDCT	Inverse Discrete Cosine Transform
INFN	Istituto Nazionale di Fisica Nucleare
JPEG	Joint Photographic Experts Group
JFFS	Journaled Flash File System
KPN	Kahn Process Network
LIBU	Line Builder
LwIP	Lightweight IP
LQCD	Lattice QCD
MDE	Model-Driven Engineering
MJPEG	Motion-JPEG
MCU	Minimum Coded Unit
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory Management Unit
MMP	Massively Multi-Processor
MPI	Message-Passing Interface
MP-SoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
OOPL	Object-Oriented Programming Language
OOP	Object-Oriented Programming
OCM	Object-Component Model
OCD	On-Chip Debugging
OS	Operating System
PDM	Program DSP Memory
POT	Peripherals-On-Tile
PSL	Platform Support Library
PN	Petri Nets
POSIX	Portable Operating System Interface [for Unix]
PThreads	POSIX Threads
QCD	Quantum Chromo-Dynamics
RISC	Reduced Instruction Set Computer
RAM	Random Access Memory
RAMDAC	RAM Digital-to-Analog Converter
ROM	Read-Only Memory
RT	Real-Time
SCDFG	Synchronous and Control Data Flow Graph
SDK	Software Development Kit
SDFG	Synchronous Data Flow Graph
SPP	Special-Purpose Processor
SoC	System-on-Chip
SLD	System-Level Design
SMP	Symmetric Multi-Processor
SRAM	Static RAM
ST-NoC	STMicroelectronics NoC

SPARC	Scalable Processor Architecture
SSC	Synchronous Serial Controller
SW	Software
TAP	Test Access Port
TCM	Tightly Coupled Memory
TG	Traffic Generator
TI	Texas Instruments
TTM	Time-To-Market
uC	Micro-Controller
USB	Universal Serial Bus
UP	Uni-Processor
unZZ	Inverse Zig-Zag Scan
VFS	Virtual File System
VLD	Variable Length Decoder
VLIW	Very Long Instruction Word
XDR	External Data Representation
XMI	XML Metadata Interchange
XML	Extended Markup Language
XUP	Xilinx Universitary Program

Bibliographie

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach : a new kernel foundation for UNIX development. In *USENIX*, pages 93–112, 1986. 3.2.1, 3.2.2

[2] T. Acharya and A. K. Ray. *Image Processing : Principles and Application.* Wiley, 2005. 1

[3] D. J. Armstrong. The quarks of object-oriented development. *Communication of the ACM*, 49(2) :123–128, 2006. 3.2.1

[4] P. J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. 3.1.3

[5] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3) :131–146, 2005. 3.1.2, 3.1.2

[6] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli. Metropolis : an integrated electronic system design environment. *IEEE Computer*, 36(4) :45–52, 2003. 2.3.2, 3.1.1

[7] P. Banerjee, U. N. Shenoy, A. N. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. G. Joisha, A. K. Jones, A. Kanhere, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM*, pages 39–48, 2000. 2.3.2

[8] J. Bernabeu-Auban, P. Hutto, M. Khalidi, M. Ahamad, W. Appelbe, P. Dagupta, R. LeBlanc, and U. Ramachandran. The architecture of ra : a kernel for Clouds. *System Sciences, 1989. Vol.II : Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, 2 :936–945 vol.2, Jan 1989. 3.2.1, 3.2.2

[9] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, pages 267–284, 1995. 3.2.1, 3.2.2

[10] O. Beucher. *MATLAB und Simulink (Scientific Computing).* Pearson Studium, 08 2006. 3.1.3

[11] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. 3.2.1

[12] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition.* O'Reilly Media, Inc., 3 edition, November 2005. 2.2.2

[13] J. Buck, E. A. Lee, and D. G. Messerschmitt. Ptolemy : a framework for simulating and prototyping heterogeneous systems. 1992. 2.3.2, 3.1.1

[14] E. Burnette. *Hello, Android : Introducing Google's Mobile Development Platform*. Pragmatic Bookshelf, 2008. 2.2.2

[15] L. Cai, D. Gajski, and M. Olivarez. Introduction of system level architecture exploration using the SpecC methodology. In *ISCAS (5)*, pages 9–12, 2001. 2.3.2

[16] R. H. Campbell and S. mong Tan. Choices : an object-oriented multimedia operating system. In *In Fifth Workshop on Hot Topics in Operating Systems, Orcas Island*, pages 90–94. IEEE Computer Society, 1995. 3.2.1, 3.2.2

[17] A. Chagoya-Garzon, X. Guérin, F. Rousseau, F. Pétrot, D. Rossetti, A. Lonardo, P. Vicini, and P. S. Paolucci. Synthesis of communication mechanisms for multi-tile systems based on heterogeneous multi-processor system-on-chips. In *IEEE/IFIP RSP*, June 23-26 2009. 4.3.2, 5.2.5

[18] M. Coppola, M. D. Grammatikakis, R. Locatelli, G. Maruccia, and L. Pieralisi. *Design of Cost-Efficient Interconnect Processing Units : Spidergon STNoC*. CRC Press, Inc., Boca Raton, FL, USA, 2008. 5.2.2

[19] L. B. de Brisolara, S.-I. Han, X. Guérin, L. Carro, R. Reis, S.-I. Chae, and A. A. Jerraya. Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC. In *SCOPES*, pages 81–89, 2007. 3.1.6

[20] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys*, 2003. 4.3.4

[21] B. K. Dwivedi, A. Kumar, and M. Balakrishnan. Automatic synthesis of system-on-chip multiprocessor architectures for process networks. In *CODES+ISSS*, pages 60–65, 2004. 2.3.2

[22] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001. 4.4.4

[23] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel : An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995. 3.2.1, 3.2.2

[24] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think : a software framework for component-based operating system kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002. 3.2.1, 3.2.2, 3.3.1

[25] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit : a substrate for kernel and language research. In *SOSP*, pages 38–51, 1997. 3.2.1, 3.2.2

[26] E. Gabber, C. Small, J. L. Bruno, J. C. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *USENIX Annual Technical Conference, General Track*, pages 267–282, 1999. 3.2.1, 3.2.2

[27] P. Gerin, X. Guérin, and F. Pétrot. Efficient implementation of native software simulation for MPSoC. In *DATE*, pages 676–681, 2008. 4.1.2

[28] P. Gerin, M. M. Hamayun, and F. Pétrot. Native MPSoC co-simulation environment for software performance estimation. In *CODES+ISSS*, pages 403–412, 2009. 5.3.4

[29] P. Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. Technical report, Xenomai, 2004. 2.2.2

[30] M. Girkar and C. D. Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *Int. J. Parallel Program.*, 22(5) :519–551, 1994. 3.1.1, 2.3, 3.1.1

[31] GNU. The ld GNU's linker manual. http://sourceware.org/binutils/docs/ld/index.html, 2010. 3.1.3, 3.1.7

[32] P. Guironnet de Massas and F. Pétrot. Comparison of memory write policies for NoC based multicore cache coherent systems. In *DATE*, pages 997–1002, 2008. 5.3.4

[33] R. Gupta. *Introduction to Lattice QCD*. Elsevier, 1999. 5.2.1

[34] S. Hacker, H. Bortman, and H. Bartman. *The BeOS Bible*. Addison Wesley Longman, 1 edition, April 22 1999. 4.2, 4.2

[35] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guérin, A. A. Jerraya, K. Huang, L. Li, and X. Yan. Simulink®-based heterogeneous multiprocessor soc design flow for mixed hardware/software refinement and simulation. *Integration, the VLSI Journal*, 2008. 2.3.2

[36] S.-I. Han, X. Guérin, S.-I. Chae, and A. A. Jerraya. Buffer memory optimization for video codec application modeled in simulink. In *DAC*, pages 689–694, 2006. 3.1.1, 3.1.6

[37] R. Harrison. *Symbian OS C++ for Mobile Phones*. Wiley, June 16 2003. 2.2.2

[38] G. Heineman and W. Councill. *Component-Based Software Engineering : Putting the Pieces Together*. Addison-Wesley Professional, 2001. 3, 3

[39] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8) :666–677, 1978. 3.1.2, 3.1.2

[40] Institute of Electrical and Electronics Engineers. *Portable Operating System Interface (POSIX) – Part 1 : System Application Program Interface*. Number 1003.1, [C89]/IEC 9945-1 in IEEE Std. Institute of Electrical and Electronics Engineers, 1990. 4.2.2, 4.3.1

[41] Institute of Electrical and Electronics Engineers. IEEE standard test access port and boundary scan architecture. Technical report, IEEE Computer Society/Test Technology, 2001. 2.1.1

[42] International Electrotechnical Commission. The IEC 61508 Safety Standard. Technical report, International Electrotechnical Commission, 2005. 2.3

[43] R. S. Janka. *Specification and Design Methodology for Real-Time Embedded Systems*. Springer, 2002. 2.3.2

[44] A. Jantsch and I. Sander. Models of computation in the design process. 2005. 3.1.1, 3.1.1

[45] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01 : Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag. 3.2.1, 3.2.2

[46] F. Kolnick. *The QNX 4 Real-time Operating System*. Basis Computer Systems, July 2000. 2.2.2

[47] J. J. Labrosse. *MicroC OS II : The Real Time Kernel*. CMP Books, June 15 2002. 2.2.2

[48] D. A. Lamb. An introduction to IDL. External Technical Report 1.2, Department of Computing and Information Science, Queen's University, November 1988. 3, 3

[49] H. Ledgard. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983. 1

[50] H. F. Ledgard. *The Little Book of Object-oriented Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 3.2.1

[51] J. Liedtke. Improving IPC by kernel design. In *SOSP*, pages 175–188, 1993. 4.2.1

[52] J. Liedtke. On micro-kernel construction. In *SOSP*, pages 237–250, 1995. 3.2.1, 3.2.2

[53] P. Lieverse, P. van der Wolf, K. A. Vissers, and E. F. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *VLSI Signal Processing*, 29(3) :197–207, 2001. 2.3.2

[54] LIP6. The SoCLib project. http://www.soclib.fr, January 2005. 5.3.3

[55] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. The design of application-tailorable operating system product lines. In *CASSIS*, pages 99–117, 2005. 3.2.1, 3.2.2

[56] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *WORDS*, pages 413–420, 2005. 3.2.1, 3.2.2

[57] D. Lohmann and O. Spinczyk. Developing embedded software product lines with aspectc++. In *OOPSLA '06 : Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 740–742, New York, NY, USA, 2006. ACM. 3.2.1, 3.2.2

[58] A. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002. 3.2.1, 4.1, 2.2.2, 3.2.2, 4.1

[59] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example : Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. 2.2.2

[60] Message Passing Interface Forum. MPI : A message-passing interface standard. 1994. 4.1.1

[61] Microsoft. Fat32 file system specification. Technical Report 1.03, Microsoft Corporation, December 2000. 4.1.2

[62] J. G. Mitchell, J. Gibbons, G. Hamilton, P. B. Kessler, Y. Y. A. Khalidi, P. Kougiouris, P. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An overview of the Spring system. In *COMPCON*, pages 122–131, 1994. 3.2.1, 4.2, 3.2.2, 4.2

[63] S. mong Tan, D. K. Raila, and R. H. Campbell. An object-oriented nano-kernel for operating system hardware support. In *In Fourth International Workshop on Object-Orientation in Operating Systems*, pages 220–223. IEEE Computer Society, 1995. 3.2.1, 3.2.2

[64] M. G. Morris, C. Speier, and J. A. Hoffer. An examination of procedural and object-oriented systems analysis methods : Does prior experience help or hinder performance ? *Decision Sciences*, 30(1) :107–136, 1999. 3.2.1

[65] J. Murray. *Inside Microsoft Windows CE*. Microsoft Press, October 2 1998. 2.2.2

[66] P. R. Panda. SystemC : a modeling platform supporting multiple design abstractions. In *ISSS '01 : Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, New York, NY, USA, 2001. ACM. 3.1.3

[67] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins. Deterministic parallel processing. *International Journal of Parallel Programming*, 34(4) :323–341, 2006. 4.4.2

[68] P. S. Paolucci, A. A. Jerraya, R. Leupers, L. Thiele, and P. Vicini. SHAPES : a tiled scalable software hardware architecture platform for embedded systems. In *CODES+ISSS*, pages 167–172. ACM, New York, NY, USA 2006. 4.4, 4.3.2

[69] C. Passerone. Real time operating system modeling in a system level design environment. In *ISCAS*, 2006. 2.3

[70] F. Pétrot and P. Gomez. Lightweight implementation of the POSIX threads API for an on-chip MIPS multiprocessor with VCI interconnect. In *DATE*, pages 20051–20056, 2003. 2.2.2

[71] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2) :99–112, 2006. 2.3.2

[72] A. D. Pimentel, P. van der Wolf, E. Deprettere, J. T. J. Van Eijndhoven, L. Hertzberger, and S. Vassiliadis. The artemis architecture workbench. 2000. 2.3.2

[73] K. Popovici, X. Guérin, F. Rousseau, P. S. Paolucci, and A. A. Jerraya. Platform-based software design flow for heterogeneous MPSoC. *ACM Transaction on Embedded Computing System*, 7(4) :1–23, 2008. 4.1.2

[74] C. Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. 4.1.2

[75] D. Probert, J. L. Bruno, and M. Karaorman. SPACE : A new approach to operating system abstraction. In *International Workshop on Object Orientation in Operating Systems*, pages 133–137, 1991. 3.2.1, 3.2.2

[76] RedHat. The newlib c library. http://sourceware.org/newlib/, 2010. 4.3.3

[77] D. Ritchie and K. Thompson. The UNIX time-sharing system. *Communication of the ACM*, 17(7) :365–375, 1974. 3.2.1, 3.2.2

[78] D. Robson. Object-oriented software systems. In *Byte 6*, pages 74–86, August 1981. 3.2.1

[79] RTCA. Software considerations in airborne systems and equipment certification. Technical Report DO-178B, RTCA, Inc., 1992. 2.3

[80] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006. 3.2.2, 3.2.3

[81] W. Schröder-Preikschat, D. Lohmann, F. Scheler, and O. Spinczyk. Dimensions of variability in embedded operating systems. *Inform., Forsch. Entwickl.*, 22(1) :5–22, 2007. 3.2.1, 4.2, 3.2.2, 4.2

[82] R. M. Soley and C. M. Stone. *Object Management Architecture Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1995. 3, 3

[83] D. A. Solomon. The windows NT kernel architecture. *Computer*, 31(10) :40–47, 1998. 3.2.1, 3.2.2

[84] M. Stefik and D. G. Bobrow. Object-oriented programming : Themes and variations. *AI Magazine*, 6(4) :40–62, 1986. 3.2.1

[85] J. E. Stone. An efficient library for parallel ray tracing and animation. Master's thesis, University of Missouri, 1998. 5.3.4

[86] Sun. External data representation standard. Technical Report RFC1014, Sun Microsystems, Inc., June 1987. 4.1.1

[87] A. S. Tanenbaum and S. J. Mullender. An overview of the Amoeba distributed operating system. *SIGOPS*, 15(3) :51–64, 1981. 3.2.1, 3.2.2

[88] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. 7th Int'l Conference on Application of Concurrency to System Design (ACSD'07)*, pages 29–40, Bratislava, Slovak Republic, 2007. 4.3.2, 5.2.3

[89] T. Thomsen, L. Köster, and R. Stracke. Connecting Simulink to OSEK : Automatic code generation for real-time operating systems with Targetlink. In *Embedded Intelligence*, February 2001. 2.3.2

[90] R. Vilbig. JTAG debug - everything you need to know. Technical report, Mentor Graphics, 2009. 2.1.1

[91] C. Wehner. *Tornado and Vxworks*. Books on Demand, 2006. 2.2.2

[92] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5) :15–31, 2007. 4.4.2

[93] R. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Commun. ACM*, 33(9) :104–124, 1990. 3.2.1

[94] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs : Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995. 5.3.4

[95] E. Yourdon, P. Nevermann, K. Oppel, J. Thomann, and K. Whitehead. *Mainstream objects : an analysis and design approach for business*. Yourdon Press, Upper Saddle River, NJ, USA, 1995. 3.2.1

Publications

[1] X. Guérin and F. Pétrot. *Multi-Core Embedded Systems*, chapter 9 — "Operating System Support for Applications targeting Multi-Core System-on-Chips". Taylor & Francis Group LLC CRC Press, 2010

[2] X. Guérin and F. Pétrot. A system framework for the design of embedded software targeting heterogeneous multi-core SoCs. In *ASAP*, pages 153–160, July 6 2009.

[3] X. Guérin, K. Popovici, W. Youssef, F. Rousseau, and A. A. Jerraya. Flexible application software generation for heterogeneous multi-processor system-on-chip. In *COMPSAC*, volume 1, pages 279–286, 2007.

[4] P. Gerin, X. Guérin, and F. Petrot. Efficient implementation of native software simulation for MPSoC. In *DATE*, pages 676–681, 2008.

[5] A. Chagoya-Garzon, X. Guérin, F. Rousseau, F. Pétrot, D. Rossetti, A. Lonardo, P. Vicini, and P. S. Paolucci. Synthesis of communication mechanisms for multi-tile systems based on heterogeneous multiprocessor system-on-chips. In *IEEE/IFIP RSP*, June 23-26 2009.

[6] L. B. de Brisolara, S.-I. Han, X. Guérin, L. Carro, R. Reis, S.-I. Chae, and A. A. Jerraya. Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC. In *SCOPES*, pages 81–89, 2007.

[7] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guérin, A. A. Jerraya, K. Huang, L. Li, and X. Yan. Simulink®-based heterogeneous multiprocessor soc design flow for mixed hardware/software refinement and simulation. *Integration, the VLSI Journal*, 2008.

[8] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, R. Reis, X. Guérin, and A. A. Jerraya. Memory-efficient multithreaded code generation from simulink for heterogeneous MPSoC. *Design Automation for Embedded Systems*, 11(4), December 2007.

[9] S.-I. Han, X. Guérin, S.-I. Chae, and A. A. Jerraya. Buffer memory optimization for video codec application modeled in simulink. In *DAC*, pages 689–694, 2006.

[10] K. Huang, S.-I. Han, K. Popovici, L. B. de Brisolara, X. Guérin, L. Li, X. Yan, S.-I. Chae, L. Carro, and A. A. Jerraya. Simulink-based MPSoC design flow Case study of motion-JPEG and H.264. In *DAC*, pages 39–42, 2007.

[11] K. Huang, X.-L. Yan, S.-i. Han, S.-I. Chae, A. A. Jerraya, K. Popovici, X. Guérin, L. Brisolara, and L. Carro. Gradual refinement for application-specific MPSoC design from simulink

model to RTL implementation. *Journal of Zhejiang University SCIENCE A*, pages 151–164, 2008.

[12] K. Popovici, X. Guérin, L. Brisolara, and A. Jerraya. Mixed hardware software multilevel modeling and simulation for multithreaded heterogeneous MPSoC. *VLSI Design, Automation and Test, 2007. VLSI-DAT 2007. International Symposium on*, pages 1–4, April 2007.

[13] K. Popovici, X. Guérin, F. Rousseau, P. S. Paolucci, and A. A. Jerraya. Efficient software development platforms for multimedia applications at different abstraction levels. In *IEEE International Workshop on Rapid System Prototyping*, pages 113–122, 2007.

[14] K. Popovici, X. Guérin, F. Rousseau, P. S. Paolucci, and A. A. Jerraya. Platform-based software design flow for heterogeneous MPSoC. *ACM Transaction on Embedded Computing System*, 7(4) 1–23, 2008.

Résumé Cette dissertation montre que des applications embarquées complexes peuvent tirer partie efficacement de plateformes MP-SoC hétérogènes tout en respectant les critères de flexibilité, mise à l'échelle, portabilité et time-to-market. Elle fait la description d'un flot de conception de logiciel embarqué amélioré combinant un générateur de code, GECKO, et un environnement logiciel innovant, APES, afin d'obtenir un haut niveau d'efficacité. La contribution ainsi présentée est double : 1) un flot de conception de logiciel embarqué amélioré avec un ensemble d'outils permettant la construction automatique d'objets binaires minimaux pour une application donnée ciblant une plateforme MP-SoC donnée, et 2) un ensemble de composants logiciels modulaire et portable incluant des mécanismes de systèmes d'exploitations traditionnels ainsi que le support de multiples processeurs.

Mots clés Système d'exploitation, approche à composants, abstraction du matériel, flot de conception, ingénierie dirigée par les modèles, MP-SoC

Abstract This dissertation shows that complex, embedded software applications can effectively operate heterogeneous MP-SoC with respect to flexibility, scalability, portability, and Time-To- Market. It presents an improved embedded software design flow that combines an application code generator, GECKO, and a novel software framework, APES, to achieve a high level of efficiency. Our contribution is twofold : 1) an improved embedded software design flow with several tools that enable the automatic construction of minimal and optimized binaries for a given application targeting a given MP-SoC, and 2) a modular and portable set of software components that includes traditional operating system mechanisms as well as the support for multiple processors.

Keywords Operating system, component-based design, hardware abstractions, design flow, model-driven engineering, MP-SoC

ISBN : 978-2-84813-154-2
Laboratoire TIMA - 46, avenue Félix Viallet, 38000 GRENOBLE