On the compilation of a parallel language targeting the self-adaptive virtual processor

Bernard, T.A.M.

Citation for published version (APA):
Computer-based systems are ubiquitous in daily life: in fridges which are regulated with an embedded Integrated Circuit (IC); in car control systems which manage the major driving features and engine controls; etc. The commodity computer market is driven by demands on delivering more efficient and reliable computing appliances at frequent intervals of time: from a few months to a few years. These computing systems have to cope with the ever-growing increase of resource-intensive applications. Perhaps the mobile phone market is the most relevant market in which to observe this demand, during the market explosion of the last two decades [1]. A cell phone has evolved from a simple voice-calling device to an advanced multi-purpose personal assistant implementing the latest standards [2]. Not only do modern cell phone devices allow users to make voice calls, they also connect to the Internet, send and receive emails, organize the user’s agenda, and much more. Also referred to as smart phones, these portable devices assist the user in her/his daily life with various applications: video calls, embedded camera, phone games, on-line chatting applications, social networking, etc. Therefore, the cell phone market hardware demands have exploded due to the pressures of having more advanced features embedded in their applications. Nonetheless, smart phones still have to deal with the physical issues inherited from traditional IC design: low battery life and additional heat after extended usage.
1.1 Classical microprocessor improvements

To allow such technology to exist, extensive research has been conducted by semiconductor companies who provide the heart of computing systems: the integrated circuit also known as the chip. The first microprocessor, or processor, emerged in the early 1970s incorporating functions of the computer’s Central Processing Unit (CPU) on a single chip [3]: arithmetic unit, on-chip memory (registers and caches), control and logical unit, clock, pipelines. For the last five decades, IC improvements have been made by the two following methods:

- increasing the processor’s clock rate and,
- increasing the number of instructions issued per cycle.

Announced in 1965 in [4], Moore’s law predicts that approximately every two years, the number of transistors on a chip doubles. Since then, the trend of processor performance has shown that the law depicts a certain reality in chip design. However, the more transistors present on a chip, the more physical issues appear such as:

- power consumption and,
- heat dissipation.

These classical problems of chip improvements are difficult to overcome, for example, in the domain of embedded systems such as smart phones design, where these problems become concrete annoyances for users. Indeed, mobile computing devices have a short battery life especially when using computing-expensive applications such as video calls, etc. Moreover, the devices expend excessive heat after long usage. Consequently, semiconductor companies have to look at different chip design approaches for future chip generations.

To summarize, demand in new computing systems persists, while the offer struggles to provide efficient solutions for relieving these computing systems of the aforementioned physical issues. These solutions focus on using conventional improvements. Therefore, there is an urgent need to find other ways for microprocessor design improvements.

1.2 Multicore architectures

Exploiting parallelism is not new. Parallel programming paradigms, architectures and languages have been researched for decades; they have existed since the 1970s and 1980s [5]. However, they have never been mainstream [6]. In the last few years though, the move to parallel architectures in mainstream computing became a reality [7]. New challenges in computing when exploiting concurrent architectures appear as reported in [8] and they have become
urgent to solve. These concurrent architectures are called Chip-MultiProcessors (CMPs) and they are introduced as a solution to save mainstream computing with its never-ending quest of better performance.

Nielsen et al.\[9\] describe the challenges and the benefits of multicore architectures in the context of High-Performance Computing (HPC). Indeed, dealing with multiple processors on a die pushes forward the issues of taking full advantage of this technology. Improving the performance of applications requires exposure of extreme levels of software parallelism. Sohi \[10\] suggests that the appearance of CMPs changed the directions of the computing research agenda. Articles \[11, 12\] present also the concerns with multicore architectures.

We think that one of the biggest problems facing the computer industry today is the challenge of programming the expected progression of Many-Core Chip-MultiProcessors (MCCMPs) that will reflect the beyond-frequency advances in computer performance due to the vestiges of Moore’s law. It is anticipated that tens or even hundreds of thousands of cores will be possible on a single chip at the end of silicon scaling. Moreover, it is well understood that power efficiency requires many simpler cores in a processor architecture rather than fewer more complex ones. However, exploiting these new architectures requires the exposure of explicit concurrency in the code they execute, in contrast to the implicit concurrency exploited in more complex cores. This in turn requires applications to expose this concurrency, either explicitly by the programmer using some concurrency model or automatically using parallelizing compilers, neither of which is easy \[13\]. This challenge is well acknowledged \[14, 15, 16\], but the concurrency revolution is happening now and urgently requires new tools and new ways of thinking \[17\].

1.3 Exploiting concurrency as a solution

1.3.1 The different levels of parallelism

The concurrency revolution is happening now in computing systems \[17\]. The impact of concurrency on software systems appears at all levels, from the application side to the architecture side. Achieving better performance using concurrency relies on finding parallelism in the algorithms of a given application. The parallelization of an algorithm into tasks is referred to Task-Level Parallelism (TLP). TLP focuses on distributing execution tasks (e.g. threads) across different parallel computing nodes. Unfortunately in some applications, it is difficult to discover parallelism, for example, an unconvertible sequential algorithm that cannot be decomposed in small functional tasks. Hence, there is no way to extract any benefit in concurrent executions of such algorithms compared to a sequential execution. Parallelism limitations of an application are expressed with Amdahl’s law: the speedup of a program using multiple computing nodes in parallel computing is limited by the sequential section of the program. Therefore, it is important that an architecture can also take advantage of a finer-
grained concurrency with operations, named *Instruction-Level Parallelism* (ILP). Thus, even if the functional parallelism of a given program is non existent, there are still ways to improve performance by performing operations simultaneously. In addition to this, another type of concurrency can be exploited with *Data Level Parallelism* (DLP). For instance with a loop (sometimes called *Loop-Level Parallelism* (LLP)), data parallelism is achieved when each computing node performs the same task on different pieces of distributed data.

### 1.3.2 Approaching concurrency with a cooking recipe example

Concurrency concepts are difficult to comprehend. For that reason, we use in this section a practical example with a *cooking recipe problem*. We assume this problem (with a large size) to grasp the essence and the significance of concurrency concepts and their relevance to performance. A recipe from a cook book is fundamentally sequential when you read it. There is a list of steps to perform to accomplish the dish which is the end result. The sequential way to process is to start from the first step of the recipe and perform it. Once this first is finished, the next one on the list can be processed until completion. And this is the same for all of the following steps until the end of the recipe. The advantage of this is that the processing is deterministic; we know at any point in the recipe what is the status of the overall execution and the outcome of a transition from one state to another is predictable. We are interested in rendering the recipe’s execution parallel while keeping it deterministic. With a large problem size, the recipe has to be produced for 100 guests in a fully-equipped kitchen with a crew of cooks and a multi-unit cooker. It thus becomes important to execute the recipe more efficiently and faster; the sequential way would take too much time when serving the dish (i.e. the result of the recipe) to everyone within the shortest time between the first and the last served.

1. Boil salted water in a pot. Once the water is boiling, cook the pasta for 10 minutes. After draining them, put some olive oil and stir.

2. For the sauce, slice garlic cloves on a board. In a pot, put some olive oil on a low-heat fire and add the sliced garlic. Mince the ham and add it to the pot. Stir regularly and add chopped basil leaves. Season the pot to your taste.

3. Once the pasta and the sauce are ready, mix both and cook them slowly for few minutes.

The ingredients are assumed to be the *data* to compute in this recipe problem. The *resources* on which to execute on are a multiple-fire cooker including a crew of cooks. From the steps, we can observe a trivial partitioning into tasks to make the recipe’s execution faster. Using a TLP approach, we can isolate the sections that can be done independently. Figure 1.1 illustrates the *partitioning* of the sequential-oriented recipe. For instance, Task 1’s goal is assimilated to
the cooking of the pasta. The big problem of making the dish is decoupled into 3 sub-problems:

1. Task 1: Cook the pasta.
2. Task 2: Prepare the sauce.
3. Task 3: Assemble and finalize the dish.

The granularity of the task to be performed is important to distinguish while dealing with parallelism. The granularity can be coarse-grained with Task 1 which is a sequence of operations: pour water in a huge pot; light up the fire on the cooker; salt the water; place the pot on the fire. The granularity can be fine-grained and closer to the atomicity of an operation. For instance, in Task 2, the ham can be minced by multiple resources (i.e. multiple cooks), in the sense that the meat is the data and using a DLP approach in this matter. The ham is separated in several portions to be taken care of individually. Moreover, it is interesting to add that another type of parallelism can be performed to extract concurrency: ILP can be used to take advantage of getting simultaneous actions within Task 1: lighting up the fire on the cooker; pouring water in the pot can be done separately.

What matters, in the end, is to be able to accomplish the execution of the recipe faster, if possible, and remain sure of a correct result comparable to the sequential way of processing the recipe. This determinism of results is a priority in parallel processing and to accomplish that further steps in defining and refining the application are needed. Thus, once the tasks have been generated, it is important to formalize the communication that can occur between tasks. For instance, Task 3 requires data from Task 1, i.e. the cooked and drained pasta, as illustrated with Figure 1.2. Therefore, this exchange of data is notified explicitly in the recipe’s definition and the cooks are thus aware of it.
Figure 1.2: Communication between concurrent tasks of a cooking recipe. Task 3 requires data from task 1 (i.e. the cooked pasta) and task 2 (i.e. the ready-to-be-added sauce). Therefore, communication channels are exposed and the flow of dependent data becomes visible. We can also see that task 1 and task 2 are independent from each other.

Moreover, it is also important to isolate the synchronization points between the different tasks of the cooking recipe. One task has been first accomplished before another dependent task can be started. Figure 1.3 shows the points of synchronization necessary for the good coordination while executing the cooking recipe. Task 3 waits for Task 1 and Task 2 to be finished before it can start its execution. Consequently, we mark the recipe with a synchronization point for Tasks 1 and 2 and we put it before Task 3.

Figure 1.3: Synchronization barriers between concurrent tasks of a cooking recipe. The diamond shapes represent the points of coordination necessary for proper execution. Without coordination between the tasks, there is no guarantee of handling inter-task dependencies properly. The Start and End are respectively the entry and exiting points of the recipe.

Once the recipe has been partitioned into ‘small’ tasks, once the inter-task communication is clearly identified and once the synchronization points are visible, the concurrency management stage becomes the last step for the recipe designer (the chef) to accomplish. Concurrency management comprises: the mapping stage, which takes care of which resource and where a task will be
performed; the scheduling considers all the inter-tasks dependencies to define and regulate the order of executions of the tasks and their operations; then the dynamic resource management deals with the availability of resources during the execution of the recipe. This is illustrated with Figure 1.4. For instance, if a cook leaves the kitchen after starting the execution of the recipe or if a cook comes in during the execution, it might ask the task to be scheduled and mapped elsewhere on the available resources.

![Figure 1.4](image)

**Figure 1.4:** Management of concurrent tasks of a cooking recipe comprises multiple steps. First, mapping specifies where a task is to execute. Scheduling, considering the dependencies of the program such as inter-task communication and synchronization points, orders the tasks to be executed. Dealing with multiple resources, which may be unknown at design time, implies a dynamic management of resources while executing the tasks.

Some execution issues can occur for instance when one of the burners breaks down; the operation being accomplished on it is therefore not achieved. The entire recipe depends on that and becomes deadlocked. This is called a resource deadlock. A deadlock can also occur when the recipe has not been well designed by the chef; during execution, it just blocks unexpectedly and the operation or task waiting for another dependent one to finish never starts. Similar to a deadlock, a livelock can also happen if the cooks waiting for the ham to proceed never actually process it by constantly changing their states. Overall nothing progresses and a livelock is present. Another problem, called a race condition, can occur when two cooks try to access the same portion of the ham for example; therefore, they compete to get it first. A way to prevent that is to set up a mutual exclusion (also called mutex) to avoid the simultaneous use of a common resource.

To conclude, the major problem faced by concurrency is to be able to express these concepts properly and correctly while designing an application (i.e. the cooking recipe). We use this example as an analogy to represent a heterogeneous multi-unit target platform (i.e. the kitchen with the multi-unit cooker, the crew of cooks and the chef). Dealing with multicore architectures raises these concurrent programming problems.
1.3.3 Concurrent software design and issues

Software design evolves and requires new ways of thinking for programmers when designing applications. Designing a concurrent application resembles what we have looked at with the cooking recipe example. Thus, there are several major notions to keep in mind and their related issues when designing a problem in a concurrent manner.

**Partitioning**  The partitioning stage of design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

**Communication**  The tasks generated by the partitioning stage are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.

**Synchronization**  In the third stage, development moves from the abstract toward the concrete. Developers revisit decisions made in the partitioning and communication stages with a view to obtaining a work-flow of the algorithm that will execute. The synchronization points are then marked to coordinate the program’s execution correctly.

**Concurrency Management**  In this stage of the parallel algorithm design process, developers specify where each task will execute, referred to as *Mapping*. This mapping problem does not arise on a single computing node or even some multiple computing nodes, such as shared-memory computers that provide automatic task scheduling. Scheduling the tasks while executing the application is also part of this stage. Moreover, dynamic resource management handles the potential problems of mapping tasks as resources become available or unavailable in the set of resources.

Developers have to deal with the issues involved with these stages. In current approaches the description and management of concurrency are not decoupled, which results in a mixture of concerns that overwhelm developers making parallel application development very difficult and error-prone. This lack of a clear separation of concerns also means a lack of appropriate high-level abstractions in both the architecture and application, which precludes portability between different platforms. Thus what currently happens is that applications are either developed targeting specific platforms or existing applications are retargeted to
a platform through a painstaking process of static application mapping and the introduction of platform-specific functionality into the application itself.

1.4 Impact of concurrency on software systems

1.4.1 On a whole system

This concurrency revolution, presented in [13, 17], has an impact on software systems. Users utilize applications without knowledge of the machinery underneath as shown on Figure 1.5. Applications are executed on the hardware; the operating system provides the interface between the software and the hardware. Having a multicore architecture requires an adapted toolchain in order to operate, comprising a concurrency-oriented operating system and applications. Users do not need to be aware of the machinery. Consequently, this toolchain must be adapted to handle the features of this new concurrent target platform. Nowadays, the major issue to cope with is the multicore programming menace which is already acknowledged by the community [14, 15, 16].

The previous section exposed the concepts of concurrency that have to be embedded into software systems. The problem is that multicore architectures exist but the tools are not yet ready.

![Figure 1.5: Overview of a standard software system. The user on top of the layers employs applications such as a game on a cell phone. The gaming application is then executed on the hardware while being interfaced by the operating system.](image)

Moreover, although new concurrent target platforms are here, old sequential target platforms and their sequential-oriented applications are still heavily used. For that reason, the software community must consider backward compatibility of old software systems with these new platforms. Execution toolchains are composed of various layers of software programs. Figure 1.5 depicts interactions between these different layers of a software system used in
the execution of a program. Based on an architecture at the bottom, a series of software programs is required to make proper usage of the hardware machinery. The Operating System (OS) layer directly provides an interface between the hardware and other software layers. The OS provides a Run-Time System (RTS) managing executions of applications. Moreover, the System Layer supplies the standards libraries for network, file system, etc. Applications can be using specific components taken from the Framework Layer. In the end, layers of software systems need to be adapted to concurrency for coming multicore architectures.

The execution toolchain of a software system is necessary on all computing systems to execute applications. To build these applications, we need look at the development toolchain used to render an executable application. Figure 1.6 instances the major tools necessary for software development.

### 1.4.2 Bridging the software and the hardware worlds

In this thesis, we look at a particular part of software system development which *bridges the software side and the hardware side*. Introducing concurrency concepts as a new paradigm requires appropriate concurrency-aware tools. Articles [19, 20] analyze the different existing parallel programming models and languages available to developers. They capture the abstraction of parallelism in these models and evaluate them. Despite the effort in programming model research, the tools are not yet ready. Figure 1.6 abstracts levels of software and hardware design and development tools. For software developers, the goal is to rely on a tool that would just do the job after expressing the tasks to be performed. In other words, developers want an easy-to-use and reliable toolchain for their development. Compilers are a major component in software development. Often developers rely on them to optimize their code for an architecture that they do not need to be completely aware. The role of the compiler is to take advantage of the targeted architecture with respect to the tasks to be accomplished within the programmed applications. Therefore, the pressure on this component (in Figure 1.6 see the central box) comes from the software side where the compiler must support all source language features, and must also understand the meaning of the program in order to get the most optimized result. Pressure also arises from the hardware side which has no idea nor concept of the software implementation constraints; the compiler must know where in the program the code can be improved and tweaked to take the greatest advantage for the targeted architecture.

Consequently, bridging these two worlds is the bottleneck where the *compiler*, the software system transforming a user-level language into a machine-level language, operates as a middle man. This middle-man tool has be to aware of both sides of this divide in order to function; the introduction of multicore architectures has already raised concerns [21]. The work in this thesis is based on an underlying compiler development for a new parallel programming language (called \(\mu\)TC, a C-like language with keyword extension and new semantics) targeting a many-core architecture (called the Microgrid). Furthermore,
this thesis describes a novel concurrent execution model - the Self-Adaptive Virtual Processor (SVP) - which has been developed by the Computer Systems Architecture group (CSA) at the university of Amsterdam. Various implementation work related to this model is also part of the CSA group’s research (aforementioned µTC and Microgrid, respectively language and hardware implementations). The main goal of SVP is to provide an operational computing system including various areas of research such as hardware emulation using cycle-accurate simulation, high-level program simulation on conventional architectures, memory simulation, concurrent language design, operating system and compilers. Having an exotic programming language (µTC in opposition to classic C-like languages) necessitates the presence of an adapted and dedicated toolchain, stress is especially put on compilers to perform valid program transformations without loss of program semantics and to improve the code to obtain as efficient execution as possible.
1.5 Contribution of this thesis

This thesis focuses on the multicore programming menace and how we tackle that by compiling the SVP model to its various implementations. Additionally, this thesis concentrates on the issues of bridging two major components of the system: the first is the language implementation as a concurrency description mechanism; the second is the architecture implementation as a multicore platform. This thesis illustrates the issue of generating efficient and correct code for this multicore architecture. Moreover, this thesis exposes the changes required and the challenges encountered to integrate native concurrency idioms and assumptions into an existing, conventional, sequential-oriented compiler.

At first glance, this work targets a specific audience of technical computer scientists involved in compilation. Nevertheless, this work also represents a reflection on the limits of existing engineering methods to tackle the challenges of dealing with concurrency. The essence of the technical contribution of this work is then used as a theory of engineering on the limitations of current computing systems and other concurrency-based systems. In the context of facing the multicore programming menace, this work becomes relevant to an audience dealing with issues of concurrency-based compilation, language design, and multicore programming issues.

1.6 Overview of this thesis

In this thesis we aim to expose the changes and the challenges facing the integration of native concurrency idioms from a concurrent execution model into an existing sequential-based system. In Chapter 2, we study the other past and current works related to parallel computing systems. In addition, we evaluate how close or how different our research is compared with this work at different levels of granularity (at the model-, the architecture-, the compiler-, the language-level). In Chapter 3, we describe a candidate to tackle the “multicore programming menace” with the Self-Adaptive Virtual Processor (SVP), an abstract concurrent execution model. This model combines fine-grained threads (concurrent composition is assumed at all levels) with dataflow synchronization between threads. As such it is a compromise between capturing maximal concurrency (i.e. dataflow) and providing efficient implementation (i.e. threads). It provides deadlock free composition and captures locality and regularity via the constraints of the model, which means it is amenable to compiler analysis and code transformation even in the absence of a specific target architecture. Also like sequential and dataflow models it provides determinism of results. We then depict two implementations derived from this SVP model: a hardware target implementation named Microgrids, and a source language implementation named $\mu$TC (pronounced ‘myoo - ti - si’). After that in Chapter 4, we discuss the bridge between these two implementations resulting in compiling from the source language $\mu$TC into the target architecture Microgrids. We also study the
differences between classical compilation and SVP compilation. Chapter 5 investigates conflicts that SVP properties have over conventional compiler optimizations. Later Chapter 6 exposes the challenges of SVP compilation with current compilation systems. Moreover, we explain the methods used to embed these concurrency concepts. Chapter 7 discusses evaluation of SVP compilation with scientific problems. Finally, Chapter 8 summarizes this thesis work as a reflection on existing engineering methods, referred to as theory of engineering, and on the future of compilers as concurrency-aware systems.