Design Considerations of a Meta-Level Optimizing Computer System

Fumihito Furukawa†, Moriyuki Saito†, Gaku Ishihara‡, Kanemitsu Ootsu‡, Takashi Yokota†, Takanobu Baba‡
† Venture Business Laboratory, Utsunomiya University
‡ Department of Information Science, Faculty of Engineering, Utsunomiya University
7–1–2 Yoto, Utsunomiya, Tochigi, 321–8585, Japan
{bunjin, saitou, gaku, kim, yokota, baba}@virgo.is.utsunomiya-u.ac.jp

1. Introduction

Today’s computer systems have been greatly accelerated by various architectural features that exploit a huge amount of transistors. In order to further improve the performance, we need to devise new architectural technologies that effectively utilize the hardware resources, because it is becoming difficult to speedup the program execution in proportion to the increase in transistor count. In order to answer this requirement, we are investigating to utilize some chip area for both dynamically and autonomously tuning of the configuration of the multiprocessor for high performance computation[1]. We consider that the configuration should be optimized statically as well as dynamically, since the dynamic change of the program’s behavior may cause a large gap between the statically provided configuration and the behavior. In particular, if we target not only numerical applications but non-numerical ones as well, the tuning contributes to the performance improvement. The dynamic and autonomous tuning may guide the system to the best configuration for the running program, and thus allow us to effectively utilize a large amount of hardware resources for a wide variety of applications.

In this paper, based on this basic idea, we consider the design and implementation issues.

2. Meta-Level Optimization

In order to implement our idea, we define a meta-level processor as opposed to a base-level one, i.e. a conventional von Neumann machine. The meta-level processor observes the behavior of the program execution from outside the base-level processor, determines the optimal configuration of the base-level processor based on its observation and then reconfigures the base-level processor. The meta-level processor acquires the execution profile of the base-level processor and produces the optimized base-level configuration. The combined meta- and base-level processors are expected to realize an autonomous optimization according to the run-time behavior of the machine.

3. Implementation Issues

In order to implement our basic idea, we considered the followings:

3.1. Reconfiguration Target

The hardware reconfiguration, such as FPGAs, can exploit the maximal parallelism from a given program. However, the hardware reconfiguration makes it difficult to utilize the high clock rate, the basic factor of the speedup. Further, the dynamic hardware reconfiguration takes a long time. A long reconfiguration time will make it difficult to attain the speedup of the total execution time.

Therefore, we introduced a software reconfiguration. In order to exploit the maximal parallelism as the hardware reconfiguration, we have adopted a thread as the unit of reconfiguration. As the variation of primitive data type is small, we consider that ALU’s of a fixed size are appropriate units of reconfiguration and that it is natural to control the ALU’s by a VLIW sequence, i.e. a thread. We have decided to provide hardware support for the fast thread-level reconfiguration.

3.2. Reconfiguration Timing

The reconfiguration can be done both statically and dynamically. The static reconfiguration can take some time. However, it cannot follow the dynamic change of the so-called phased behaviors. The dynamic reconfiguration can follow such a change. However, it should be done quickly to avoid an increase in run-time overhead.

The selection of static or dynamic reconfiguration depends on the characteristics of the programs. If the run-time behaviors are fixed, and a single configuration fits the fixed behavior, the static reconfiguration based on the training run will be an appropriate solution. Examples of this category may be mostly simple numeric computations that treat uniform and regular data structure.

If the run-time behaviors change from input data or program execution phases, the dynamic reconfiguration will work. Here, the dynamic reconfiguration ranges from a selection of statically prepared configurations to a fully run-time reconfiguration.

In order to attain the best performance for each of a wide variety of applications, we have decided to provide both a static and dynamic reconfiguration capability.

3.3. Hardware Structure

The meta- and base-level processors can be either separate heterogeneous or unified homogeneous processors. Our initial idea was to design a specialized processor for the meta-level. We thought the hardware cost of the meta-level would be small, as this specialized processor does not contribute to the real computation. The drawback is not only its hardware overhead, but also its inflexibility caused by the hard border line between the meta- and base-levels. Sometimes, the meta-level processor may just poll the condition to start the optimization. The base-level may have nothing to do while the meta-level processor is involved in the optimization.

In order to make the hardware structure regular and in order to best utilize the resources by dynamic load balancing, we decided to employ a unified architecture, where every processor can act as either a meta- or a base-level processor. While this requires each processor to have extra hardware for acting as a meta-level processor, it does enable a flexible reconfiguration. By distributing most of the meta-level opera-
4. Execution Model

4.1. Thread-Level Reconfiguration

As described in 3.1, we have selected a thread as the unit of reconfiguration. The threads are categorized into four types: a computing thread (CT), a profiling thread (PT), an optimizing thread (OT), and a management thread (MT). The CT works at the base-level, and the PT, OT and MT work at the meta-level.

The CT executes an application code. The PT processes the profiling information to be passed to the OT while the application code is executed. The processed data is sent to the OT for optimizing the CT. The MT allocates the resources to the threads, and restarts their execution.

As described in 3.3, we made the thread units uniform. In other words, each thread unit can execute any of the four types of threads. By this strategy, we can allocate an appropriate number of threads to both meta- and base-levels, depending on the state of the execution.

4.2. Execution Control

Figure 1 illustrates how the execution proceeds. It includes the two extreme phases: the profiling-centric and the computing-centric.

The profiling-centric phase appears when the application code is first executed, and there is little information about the program. In this situation, the PTs play an important role in obtaining detailed information about the program. After the profiling information is collected, the OTs are invoked under the control of the MT. The OTs determine the optimal form of execution, and generate the optimized CT code.

In the computing-centric phase, the CTS perform a computation. In order to attain a high performance, the basic configuration of CT implements to exploit the thread level parallelism. When the optimization restart condition is satisfied, the system’s behavior changes to the profiling-centric phase.

5. System Implementation

The YAWARA system [1] is an implementation of the computation model, described above. It consists of the software and hardware systems, as follows:

The YAWARA system includes the two extreme phases: the profiling-centric and the computing-centric. In the profiling-centric phase, the profiling thread (PT) processes the profiling information to be passed to the OT while the application code is executed. The processed data is sent to the OT for optimizing the CT. The MT allocates the resources to the threads, and restarts their execution. As described in 3.3, we made the thread units uniform. In other words, each thread unit can execute any of the four types of threads. By this strategy, we can allocate an appropriate number of threads to both meta- and base-levels, depending on the state of the execution.

Figure 1 illustrates how the execution proceeds. It includes the two extreme phases: the profiling-centric and the computing-centric.

The profiling-centric phase appears when the application code is first executed, and there is little information about the program. In this situation, the PTs play an important role in obtaining detailed information about the program. After the profiling information is collected, the OTs are invoked under the control of the MT. The OTs determine the optimal form of execution, and generate the optimized CT code.

In the computing-centric phase, the CTS perform a computation. In order to attain a high performance, the basic configuration of CT implements to exploit the thread level parallelism. When the optimization restart condition is satisfied, the system’s behavior changes to the profiling-centric phase.

Figure 2 shows the organization of the software system. It consists of a static optimization system (SOS) and a dynamic optimization system (DOS). The SOS is much like a conventional compiler. If the final results of an execution profile are available, it utilizes the profile to optimize the code. The DOS performs a dynamic optimization, utilizing run-time profile information. As a part of DOS, the PT, OT and MT reflect the optimized configuration to the real hardware.

Figure 3 shows the configuration of the hardware. The basic configuration of the hardware is a chip multiprocessor. Each processor is called a Thread Engine (TE). The features of each TE are: a thread code cache, a profiling unit. Depending on the state of the execution, there can be four types of threads. By this strategy, we can allocate an appropriate number of threads to both meta- and base-levels, depending on the state of the execution.