Runtime Code Parallelization for On-Chip Multiprocessors*

M. Kandemir, W. Zhang
CSE Department
Penn State University
University Park, PA, 16802

M. Karakoy
Department of Computing
Imperial College
London SW7 2AZ, UK

Abstract

Chip multiprocessing (or multiprocessor system-on-a-chip) is a technique that combines two or more processor cores on a single piece of silicon to enhance computing performance. An important problem to be addressed in executing applications on an on-chip multiprocessor environment is to select the most suitable number of processors to use for a given objective function (e.g., minimizing execution time or energy-delay product) under multiple constraints. Previous research proposed an ILP-based solution to this problem that is based on exhaustive evaluation of each nest under all possible processor sizes. In this paper, we take a different approach and propose a pure runtime strategy for determining the best number of processors to use at runtime. This approach is more general than static techniques and can be applicable in situations where the latter cannot be.

1. Introduction and Motivation

Several research groups agree that chip multiprocessing is the next big thing in CPU design [5]. The performance improvements obtained through better process technology, better micro-architectures and better compilers seem to saturate; in this respect, on-chip multiprocessing provides an important alternative for obtaining better performance/energy behavior than what is achievable using current superscalar and VLIW architectures.

An important problem to be addressed in executing applications on an on-chip multiprocessor environment is to select the most suitable number of processors to use. This is because in most cases using all available processors to execute a given code segment may not be the best choice. There might be several reasons for that. First, in cases where loop bounds are very small, parallelization may not be a good idea at the first place as creating individual threads of control and synchronizing them may itself take significant amount of time, offsetting the benefits from parallelization. Second, in many cases, data dependences impose some restrictions on parallel execution of loop iterations; in such cases, the best results may be obtained by using a specific number of processors. Third, data communication/synchronization between concurrently executing processors can easily offset the benefits coming from parallel execution.

As a result, optimizing compiler community invested some effort on determining the most suitable number of processors to use in executing loop nests (e.g., [8]). In the area of embedded computing, the situation is even more challenging. This is because, unlike traditional high-end computing, in embedded computing one might have different parameters (i.e., objective functions) to optimize. For example, a strategy may try to optimize energy consumption under an execution time bound. Another strategy may try to reduce the size of the generated code under both energy and performance constraints. Consequently, selecting the most appropriate number of processors to use becomes a much more challenging problem. On top of this, if the application being optimized consists of multiple loop nests, each loop nest can demand a different number of processors to generate the best result. Note that if we use fewer number of processors (than available) to execute a given program fragment, the unused processors can be turned off to save energy.

Previous research (e.g., [7]) proposed a solution to this problem that is based on exhaustive evaluation of each nest under all possible processor sizes. Then, an integer linear programming (ILP) based approach was used to determine the most suitable number of processors for each nest under a given objective function and multiple energy/performance constraints. This approach has three major drawbacks. First, exhaustively evaluating each alternative processor size for each loop can be very time consuming. Second, since all evaluations are performed statically, it is impossible to take runtime-specific constraints into account (e.g., a variable whose value is known only at runtime). Third, it is not portable across different on-chip multiprocessor platforms. Specifically, since all evaluations are done under specific system parameters, moving to a different hardware would necessitate repeating the entire process.

In this paper, we take a different approach and propose a pure runtime strategy for determining the best number of processors to use at runtime. The idea is, for each loop nest,

*This work was supported in part by NSF CAREER Award #0093082.

to use the first couple of iterations of the loop to determine the best number of processors to use, and when this number is found, execute the remaining iterations using this size. Obviously, this approach spends some extra cycles and energy at runtime to determine the best number of processors to use; however, this overhead is expected to be compensated for when the remaining iterations are executed. This is particularly true for many image/video applications in embedded domain where multiple loops with large iteration counts operate on large images/video sequences. Obviously, the main advantage of this approach is that it requires little help from compiler and it can take runtime parameters into account.

In this paper, we focus on array-intensive embedded applications and make the following contributions:

- We present a runtime loop parallelization strategy for on-chip multiprocessors. This strategy uses the initial iterations of a given loop to determine the best number of processors to employ in executing the remaining iterations.
- We discuss how its behavior can be improved by being more aggressive in determining the best number of processors.
- We argue that the overheads associated with our approach can be reduced using past history information about loop executions.

It should be noted that an on-chip multiprocessor has, in general, fewer functional units per processor than a conventional superscalar or VLIW machine; therefore, the chances are higher that the functional units will perform useful work at a given time. Also, since different pieces of the application can be executed in parallel, we can expect a better throughput from an on-chip multiprocessor (as compared to a single processor system with multiple functional units). In addition, as compared to a more complex uniprocessor architecture, using a smaller, less complex processors in a multiprocessor chip is much easier; this allows us to reduce design and verification times. Simpler processors also enable system designers to use higher frequencies. Because of these reasons, we believe that on-chip multiprocessors will be used very widely in the future.

The remainder of this paper discusses our approach in detail. We are aware of previous research that used runtime adaptability for scaling on-chip memory/cache sizes (e.g., [1]) and issue widths (e.g., [6]) among other processor resources. However, to the best of our knowledge, this is the first study that employs runtime resource adaptability for on-chip multiprocessors. Recently, there have been several efforts for obtaining accurate energy behavior for on-chip multiprocessing and communication (e.g., see [4] and the references therein). These studies are complementary to the approach discussed in this paper, and our work can benefit from accurate energy estimations for multiprocessor architectures.

The rest of this paper is organized as follows. Section 2 presents our on-chip multiprocessor architecture and gives the outline of our execution/parallelization strategy. Section 3 explains our runtime parallelization approach in detail. Section 4 presents our concluding remarks.

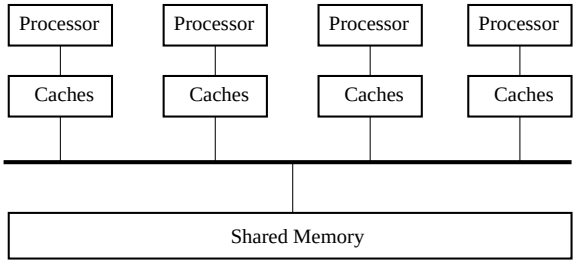

Figure 1. Relevant parts of our on-chip multiprocessor.

2. On-Chip Multiprocessor and Code Parallelization

Figure 1 shows the important components of the architecture assumed in this research. Each processor is equipped with data and instruction caches and can operate independently; i.e., it does not need to synchronize its execution with those of other processors unless it is necessary. There is also a global (shared) memory through which all data communication is performed. Also, processors can use the shared bus to synchronize with each other when such a synchronization is required. In addition to the components shown in this figure, the on-chip multiprocessor also accommodates special-purpose circuitry, clocking circuitry, and I/O devices. In this work, we focus on processors, data and instruction caches, and the shared memory.

The scope of our work is array-intensive embedded applications. Such applications frequently occur in image and video processing. An important characteristic of these applications is that they are loop-based; that is, they are composed of a series of loop nests operating on large arrays of signals. In many cases, their access patterns can be statically analyzed by a compiler and modified for improved data locality and parallelism. An important advantage of on-chip multiprocessing from the software perspective is that such an architecture is very well suited for high-level parallelism; that is, the parallelism that can be exploited at the source level (loop level) using an optimizing compiler. In contrast, the parallelism that can be exploited by single processor superscalar and VLIW machines are low level (instruction level). Previous compiler research from scientific community reveals that array-intensive applications can be best optimized using loop-level parallelization techniques [10]. Therefore, we believe that array-intensive embedded applications can get the most benefit from an on-chip multiprocessor.

An array-intensive embedded application can be executed on this architecture by parallelizing its loops. Specif-

Minimize execution time	$min(X)$
Minimize total energy	$min(\sum E_j)$
Minimize energy of component i under performance constraint	$min(E_i)$ under $X \leq X_{max}$
Minimize execution time under energy constraint for component i	$min(X)$ under $E_i \leq E_{max}$
Minimize energy-delay product	$min(X \sum E_j)$

Figure 2. Different compilation strategies (note that this is not an exhaustive list).

ically, each loop is parallelized such that its iterations are distributed across processors. An effective parallelization strategy should minimize the inter-processor data communication and synchronization. In other words, ideally, each processor should be able to execute independently without synchronization or communication. However, as mentioned earlier, in many cases, data dependences that occur across loop iterations prevent synchronization-free execution. In addition to effective parallelization, an equally important issue that affects the behavior of the application is data locality. Since each processor has its private data cache, it is very important that most of the time it finds the requested data item in its cache. Going to the large shared memory can be very costly from both the execution cycles and energy consumption perspectives.

There are different ways of parallelizing a given loop nest (see Wolfe's book [10]). A parallelization strategy can be oriented to exploiting the highest degree of parallelism (i.e., parallelizing as many loops as possible in a given nest), achieving load balance (i.e., minimizing the idle processor time), achieving good data locality (i.e., making effective use of data cache), or a combination of these. Since we are focusing on embedded applications, the objective functions that we consider are different from those considered in general purpose parallelization. Specifically, we focus on the following type of compilation strategies:

$$\text{minimize } f(E_1, E_2, \cdots, E_k, X) \text{ under } g_i(E_1, E_2, \cdots, E_k, X).$$

Here, k is the number of components we consider (e.g., processor, caches, memory). E_j is the energy consumption for the jth component and X is the execution time. $f(.)$ is the objective function for the compilation and each $g_i(.)$ where $1 \leq i \leq C$ denotes a constraint to be satisfied by the generated output code. Classical compilation objectives such as minimizing execution time or minimizing total energy consumption can easily be fit into this generic compilation strategy. Figure 2 shows how several compilation strategies can be expressed using this approach. Since we parallelize each loop nest in isolation, we apply the compilation strategy given above to each nest separately.

3. Runtime Parallelization

3.1. Approach

Let $\mathcal{I}$ be the set of iterations for a given nest that we want to parallelize. We use $\mathcal{I}' \in \mathcal{I}$ (a subset of $\mathcal{I}$) for determining the number of processors to use in executing the iterations in set $\mathcal{I} - \mathcal{I}'$. The set $\mathcal{I}'$, called the *training set*, should be very small in size as compared to the iteration set, $\mathcal{I}$. Suppose that we have K different processor sizes. In this case, the training set is divided into K subsets, each of which containing $\mathcal{I}'/K$ iterations (assuming that K divides $\mathcal{I}'$ evenly).

Let f_k denote the processor size used for the k^{th} trial, where $1 \leq k \leq K$. We use $T_{f_k}(\mathcal{J})$ to express the execution time of executing a set of iterations denoted by $\mathcal{J}$ using f_k. Now, the original execution time of a loop with iteration set $\mathcal{I}$ using a single processor can be expressed as $T_1(\mathcal{I})$. Applying our training-based runtime strategy gives an execution time of

$$T_{all} = \sum_{k=1}^{K} T_{f_k}(\mathcal{I}'/K) + T_{f_{kbest}}(\mathcal{I} - \mathcal{I}').$$

In this expression, the first component gives the training time and the second component gives the execution time of the remaining iterations with the best number of processors selected by the training phase (denoted f_{kbest}). Consequently, $T_{f_{kbest}}(\mathcal{I}) - T_{all}$ is the extra time incurred by our approach compared to the best execution time possible. If successful, our approach reduces this difference to minimum.

Our approach can also be used for objectives other than minimizing execution time. For example, we can try to minimize energy consumption or energy-delay product. As an example, let $E_{f_k}(\mathcal{J})$ be the energy consumption of executing a set of iterations denoted by $\mathcal{J}$ using f_k processors. In this case, $E_1(\mathcal{I})$ gives the energy consumption of the nest with iteration set $\mathcal{I}$ when a single processor is used. On the other hand, applying our strategy gives an energy consumption of

$$E_{all} = \sum_{k=1}^{K} E_{f_k}(\mathcal{I}'/K) + E_{f_{kbest'}}(\mathcal{I} - \mathcal{I}').$$

The second component in this expression gives the energy consumption of the remaining iterations with the best number of processors (denoted $f_{kbest'}$) found in the training phase. Note that in general $f_{kbest'}$ can be different from f_{kbest}.

Figure 3 illustrates this training period based loop parallelization strategy. It should also be noted that we are making an important assumption here. We are assuming that the best processor size does not change during the execution of the entire loop. Our experience with array-intensive embedded applications indicates that in majority of the cases, this assumption is valid. However, there exist also cases where it

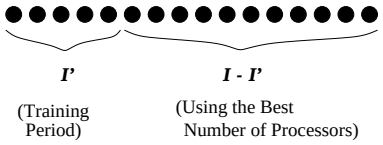

Figure 3. Parallelization based on training. Each dot represents an iteration.

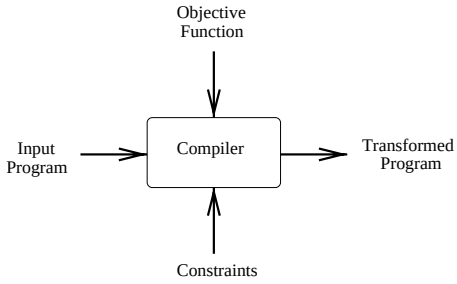

Figure 4. Inputs and output of our compiler.

is not. In such cases, our approach can be slightly modified as follows. Instead of having only a single training period at the beginning of the loop, we can have multiple training periods interspersed across loop execution. This allows the compiler to tune the number of processors at regular intervals, taking into account the dynamic variations during loop execution. The idea is discussed in more detail in Section 3.5.

An important parameter in our approach is the size of the training set, $\mathcal{I}'$. Obviously, if we have K different processor sizes, then the size of $\mathcal{I}'$ should be at least K. In general, larger the size of $\mathcal{I}'$, better estimates we can derive (as we can capture the impact of cross-loop data dependences). However, as the size of $\mathcal{I}'$ gets larger, so does the time spent in training (note that we do not want to spend too many cycles/too much energy during the training period). Therefore, there should be an optimum size for $\mathcal{I}'$, and this size depends strongly on the loop nest being optimized.

3.2. Hardware Support

In order to evaluate a given number of processors during the training period, we need some help from the hardware. Typically, the constraints that we focus on involve both performance and energy. Consequently, we need to be able to get a quick estimation of both execution cycles and energy consumption at runtime (during execution). To do this, we assume that the hardware provides a set of performance counters. These counters exist in several processors (e.g., IBM Power2, MIPS, and Pentium) for counting various types of events, such as the number of cycles, cache misses, memory coherence operations, branch mispredictions, and several categories of issued and graduated instructions. They are very useful to application developers for gaining insight into application performance and for pinpointing performance bottlenecks. As mentioned in [12], in addition to application tuning, counters have many other uses, including analyzing architectural tradeoffs, generating address traces or address statistics, evaluating compiler-based code/data transformations, and characterizing workloads. With the proliferation of dynamic compilation techniques, we can expect that most of the future processors will provide large sets of performance counters that can be used by application programmers and/or compilers.

In our approach, we employ these performance counters for two different purposes. First, they are used to estimate the performance for a given number of processors. For example, by sampling the counter that holds the number of cycles at the end of each trial (during the training period), our approach can compare the performances of different processor sizes. Second, we exploit these counters to estimate the energy consumption on different components of the architecture. More specifically, we obtain from these counters the number of accesses (for a given processor size) to the components of interest, and multiply these numbers by the corresponding per access energy costs. For instance, by obtaining the number of L1 hits and misses and using per access and per miss energy costs, the compiler calculates the energy spent in L1 hits and misses. To sum up, we adopt an activity based energy calculation strategy using the hardware counters available in the architecture.

3.3. Compiler Support

In order to experiment with different processor sizes (in the training period), we need to generate different versions of each loop at compile time. Our compiler takes the input code, constraints, and the objective function as input and generates the transformed code as output (see Figure 4). Figure 5 illustrates how a given loop nest is transformed by our compiler. In the transformed code, the first nest iterates over each processor size and records performance/energy data. After each processor size is tried, the energy and performance calculations are done. To do this, first, the counters are sampled and then the calculations to evaluate the objective function and constraints (if any) are performed. After this, a calculation (comparison) is performed to determine the best processor size. Then, the second nest (i.e., the remaining iterations) executes with this best processor size. In Figure 5, `parfor` indicates a parallel for-loop; i.e., a for-loop that will be executed by multiple processors.

3.4. Overheads

When one looks at the transformed code given in Figure 5, it is easy to see that there are several overheads that will be incurred at runtime. In this subsection, we discuss these overheads.

```
for(i=LB;i<UB;i++)
  {
    ...
  }

           ⇓

for(p=1;p<K+1;p++)
  {
    /* initialize counters */
    parfor(i=LB*p;i<LB*(p+1);i++)
      {
      /* Execute the loop parallel using
         fp processors */
        ...
      }
    /* sample the counters */
    /* perform energy/performance
       calculations */
  }
/* select the best number of
   processors (fbest) */
parfor(i=LB*(K+1);i<UB;i++)
  {
  /* Execute the loop parallel using
     fbest processors */
    ...
  }
```

Figure 5. Original and compiler transformed loops.

3.4.1 Sampling Counters

The architectures that provide performance counters also provide special instructions to read (sample) and initialize them. We assume the existence of such instructions that can be invoked from both C and assembly codes. Executing these instructions consume both energy and execution cycles in processor datapath, instruction cache, and memory.

3.4.2 Energy/Performance Calculations

As mentioned earlier, after trying each processor size, we need to compute the objective function and constraints. Note that this calculation needs to be done at runtime. To reduce the overhead of these computations, we first calculate constraints since if any of the constraints is not satisfied we do not need to compute the objective function. After all trials have been done, we compare the values of the objective functions (across all processor sizes experimented) and select the best number of processors. Our implementation also accounts for energy and execution cycle costs of these runtime calculations.

3.4.3 Activating/Deactivating Processors

During the training phase we execute loop iterations using different number of processors. Since re-activating a processor which is not in active state takes some amount of time as well as energy, we start trying different number of processors with the highest number of processors (i.e., the largest possible processor size). This helps us avoid processor re-activation during the training period. However, when we exit the training period and move to start executing the remaining iterations with the best number of processors, we might need to re-activate some processors. An alternative approach would be not turning off processors during the training period. In this way, no processor re-activation is necessary; the downside is some extra energy consumption. It should be noted, however, when we move from one nest to another, we might still need to re-activate some processors as different nests might demand different processor sizes for the best results.

3.4.4 Locality Issues

Training periods might have another negative impact on performance too. Since the contents of data caches are mainly determined by the number of processors used and their access patterns, frequently changing the number of processors used for executing the loop can distort data cache locality. For example, at the end of the training period, one of the caches (the one whose corresponding processor is used to measure the performance and energy behavior in the case of one processor) will keep most of the current working set. If the remaining iterations need to reuse these data, they need to be transferred to their caches, during which data locality may be poor.

3.5. Conservative Training

So far, we have assumed that there is only a single training period for each nest during which all processor sizes are tried. In some applications, however, the data access pattern and performance behavior can change during the loop execution. If this happens, then the best number of processors determined using the training period may not be valid anymore. Our solution is to employ multiple training sessions. In other words, from time to time (i.e., at regular intervals), we run a training session and determine the number of processors to use until the next training session arrives. This strategy is called the conservative training (as we conservatively assume that the best number of processors can change during the execution of the nest).

If minimizing execution time is our objective, under conservative training, the total execution time of a given nest is

$$T_{all} = \sum_{m=1}^{M} [\sum_{k=1}^{K} T_{f_{m,k}}(\mathcal{I}'_{m,k}) + T_{f_{m,best}}(\mathcal{I}''_m)].$$

Here, we assumed a total of M training periods. In this formulation, $f_{m,k}$ is the number of processors tried in the k^{th} trial of the m^{th} training period (where $1 \leq m \leq M$). $\mathcal{I}'_{m,k}$ is the set of iterations used in the k^{th} trial of the m^{th} training period and $\mathcal{I}''_m$ is the set of iterations executed with the best number of processors (denoted $f_{m,best}$) determined

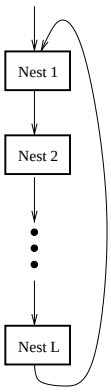

Figure 6. A program structure with nest reuse.

by the m^{th} training period. It should be observed that

$$\sum_{m=1}^{M}\sum_{k=1}^{K}\mathcal{I}'_{m,k} + \sum_{m=1}^{M}\mathcal{I}''_{m} = \mathcal{I},$$

where $\mathcal{I}$ is the set of iterations in the nest in question. Similar formulations can be given for energy consumption and energy-delay product as well. When we are using conservative training, there is an optimization the compiler can apply. If two successive training periods generate the same number (as the best number of processors to use), we can optimistically assume that the loop access pattern is stabilized (and the best number of processors will not change anymore) and execute the remaining loop iterations using that number. Since a straightforward application of conservative training can have a significant energy and performance overhead, this optimization should be applied with care.

3.6. Exploiting History Information

In many array-intensive applications from the embedded image/video processing domain, a given nest is visited multiple times. Figure 6 illustrates such an example scenario where L different nests are accessed within an outermost loop (e.g., a timing loop that iterates a fixed number of iterations and/or until a condition is satisfied). In most of these cases, the best processor size determined for a given nest in one visit is still valid in subsequent visits. That is, we may not need to run the training period in these visits. In other words, by utilizing the past history information, we can eliminate most of the overheads due to running the training periods.

It should be stressed that, in order to apply this optimization, it is not necessary that all the nests in the application are visited multiple times. We can simply apply it to the nests visited multiple times. Also, to be on the conservative side, after some number of visits, we can run a training period to see whether there is any change in the best number of processors.

4. Concluding Remarks

On-chip multiprocessing is an attempt to speedup applications by exploiting inherent parallelism in them. In this paper, we have made two major contributions. First, we have presented a runtime loop parallelization strategy for on-chip multiprocessors. This strategy uses the initial iterations of a given loop to determine the best number of processors to employ in executing the remaining iterations. Second, we have discussed how the overheads associated with our approach can be reduced using the past history information about loop executions.

References

[1] R. Balasubramonian et al. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In Proceedings of 33rd International Symposium on Micro-architecture, pp. 245–257, December 2000.

[2] L. Benini and G. De Micheli. System-Level Power Optimization: Techniques and Tools. ACM Transactions on Design Automation of Electronic Systems, 5(2), pp. 115–192, April 2000.

[3] A. Chandrakasan et al. Design of High-Performance Microprocessor Circuits, IEEE Press, 2001.

[4] DAC'02 Sessions: "Design Methodologies Meet Network Applications" and "System on Chip Design", New Orleans, LA, June 2002.

[5] http://industry.java.sun.com/javanews/stories/story2/0, 1072,32080,00.html

[6] A. Iyer and D. Marculescu. Power-Aware Microarchitecture Resource Scaling. In Proceedings of IEEE Design, Automation, and Test in Europe Conference, Munich, Germany, March 2001.

[7] I. Kadayif et al. An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors. In Proceedings of Design Automation Conference, New Orleans, LA, June 2002.

[8] K. S. McKinley. Automatic and Interactive Parallelization, PhD Dissertation, Rice University, Houston, TX, April 1992.

[9] S. Wilton and N. P. Jouppi. CACTI: An Enhanced Cycle Access and Cycle Time Model. IEEE Journal of Solid-State Circuits, pp. 677–687, 1996.

[10] M. Wolfe. High Performance Compilers for Parallel Computing, Addison-Wesley Publishing Company, 1996.

[11] W. Ye et al. The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool. In Proc. the 37th Design Automation Conference, Los Angeles, CA, June, 2000.

[12] M. Zagha et al. Performance Analysis Using the MIPS R10000 Performance Counters. In Proceedings of Supercomputing, Pittsburgh, November, 1996.