An Energy Saving Strategy Based on Adaptive Loop Parallelization[*]

I. Kadayif
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kadayif@cse.psu.edu

M. Kandemir
Microsystems Design Lab
Pennsylvania State University
University Park, PA 16802
kandemir@cse.psu.edu

M. Karakoy
Department of Computing
Imperial College
London SW7 2BZ, UK
karakoy@doc.ic.ac.uk

ABSTRACT

In this paper, we evaluate an adaptive loop parallelization strategy (i.e., a strategy that allows each loop nest to execute using different number of processors if doing so is beneficial) and measure the potential energy savings when unused processors during execution of a nested loop in a multi-processor on-a-chip (MPoC) are shut down (i.e., placed into a power-down or sleep state). Our results show that shutting down unused processors can lead to as much as 67% energy savings with up to 17% performance loss in a set of array-intensive applications. We also discuss and evaluate a processor pre-activation strategy based on compile-time analysis of nested loops. Based on our experiments, we conclude that an adaptive loop parallelization strategy combined with idle processor shut-down and pre-activation can be very effective in reducing energy consumption without increasing execution time.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Design, Experimentation, Performance

Keywords

Adaptive Parallelization, Multiprocessing, Energy Consumption

1. INTRODUCTION AND MOTIVATION

The computer industry's remarkable success in squeezing ever-more transistors into an ever-smaller area of silicon is tremendously increasing the computational abilities of electronic devices. As a result, very complex functions can be performed in a single chip. Multi-Processor-on-a-Chip (MPoC) is a single chip architecture that contains multiple processors with some amount of on-chip memory (SRAM), synchronization logic, I/O, and interconnect. Unlike a multi-PU (multi-processing unit) machine where the processing units operate in synchrony, in an MPoC, each processor can operate independently from other processors and synchronization is necessary only when processors share data. In fact, in executing a given application, we may want to use only a subset of the available processors if doing so is beneficial for some purpose. The remaining processors can stay idle or can even be used for executing some other application.

Note that while VLIW/superscalar processors may provide a certain level of parallelism, as indicated by Verbauwhede and Nicol [13], they are not scalable to provide high levels of performance needed by future applications, particularly those in next-generation wireless environments. On top of this, the power consumed by these architectures does not scale linearly as the number of execution units is increased. This is due to complexity of instruction dispatch unit, instruction issue unit, and large register files. Recently, automatic loop parallelization technology developed for array-intensive applications has been shown to be very effective [15]. We believe that array-intensive embedded applications can also take advantage of this technology and derive significant performance benefits from the on-chip parallelism and low-latency synchronization provided by an MPoC.

Energy concerns are becoming increasingly pressing in embedded system design. Particularly, the proliferation of embedded devices raised battery energy consumption to a first-class status in system design [3]. Note that dynamic (switching) energy consumption in an MPoC-based architecture is determined largely by the number of executing processors, whereas the static (leakage) energy depends on the number of powered-on processors. A given MPoC can consume a significant amount of energy and optimizing its energy consumption will be even more important in the future.

Adaptive parallelization is a compiler-directed optimization technique that tunes the number of processors for each part of the code according to its inherent parallelism. For example, intrinsic data dependences in a given nested loop may prevent us from using all processors. In such a case, trying to use more processors (than necessary) can lead to an increase in execution time due to increased inter-processor communication/synchronization costs. Similarly, a small loop bound may also suggest the use of fewer processor (than available) to execute a given loop. Loops in particular present an excellent optimization scope for adaptive parallelization. Since in general each loop might require a different number of processors to achieve the best performance, it might be useful to change the number of processors across the loops. Previous research on large scale parallel machines [4] reports that adaptive loop parallelization (that is, executing each loop with the best number of processors instead of fixing the number of active processors throughout the entire life of application) can be effective in maximizing the utilization of processors.

When adaptive loop parallelization is employed in a given MPoC, the unused (idle) processors can be shut down to conserve energy. Depending on the inherent degree of parallelism in different nests of a given code, such a strategy can lead to significant savings in energy. This is because shutting down a processor reduces its dynamic and leakage energy. However, one has to pay a *re-synchronization penalty* when a processor placed into a power-down (sleep)

[*]This work was supported in part by NSF Career Award #0093082.

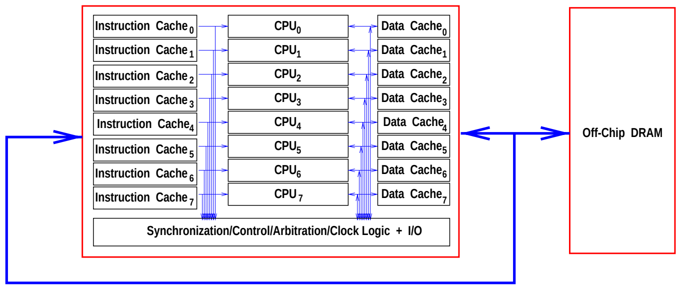

Figure 1: MPoC architecture.

state is requested to participate computation (e.g., in the next nest). The magnitude of this cost depends on the time it takes to bring the processor back from the power-down state. As will be discussed in this paper, in some cases, it might be useful to *pre-activate* a processor before it is actually needed so as to ensure that it is ready when it is required to perform computation. Such a pre-activation strategy can, if successful, eliminate the performance penalty due to re-synchronization and reduce energy consumption further.

This paper makes the following contributions:

•It evaluates an adaptive parallelization strategy and measures the potential energy savings when unused processors in the MPoC are shut down. Our results show that shutting down unused processors can lead to as much as 67% energy savings with up to 17% performance loss in a set of array-intensive applications.

• It presents a processor pre-activation strategy based on compile-time analysis of nested loops. Based on our experiments, we conclude that an adaptive loop parallelization strategy combined with idle processor shut-down and pre-activation can be very effective in reducing energy consumption without increasing execution time. Our experiments with pre-activation indicate a 39% reduction in energy consumption (on average) as compared to a scheme without energy management.

Apart from academic interest ([11, 9]), chip multiprocessor architectures are also finding their ways into commercial products. For example, Sun's MAJC-5200 [10] is a general-purpose multiprocessor system on a chip integrating two processors, a memory controller, a PCI controller, two high bandwidth I/O controllers, a data transfer engine, and a crossbar interfacing all the blocks. It is suitable for multimedia computing and networking applications. Sun Microsystems, IBM, and others are starting to produce systems that take advantage of chip multiprocessing [7]. In fact, early tests are showing that two processors in a single module outperform multiple discrete processors by 50% or more. By putting two processors on a single piece of silicon, engineers are taking advantage of shorter distances and faster bus speeds.

The remainder of this paper is organized as follows. The next section discusses the MPoC architecture assumed in this paper. Section 3 introduces our experimental platform and the array-intensive codes used in this study, and presents our experimental methodology. Section 4 discusses an adaptive parallelization strategy and presents the energy savings and performance penalties due to shutting down unused processors. Section 5 presents a processor pre-activation strategy and gives empirical data showing its effectiveness in saving energy. Section 6 discusses future work, and finally, Section 7 offers our conclusions.

2. MPOC ARCHITECTURE

We assume that a number of processors are integrated on a single die using an on-chip cache memory architecture as shown in Figure 1. In this work, we assume that each processor has a simple pipelined architecture that issues one instruction per cycle. As stated in [11], using simple, identical processors allows the design and verification costs for a single CPU core to be lower, and amortizes those costs over a larger number of processor cores. We assume that each processor has its own data cache and instruction cache. While in this work we focus on a single level on-chip data cache memory, it is straightforward to extend this architecture to designs that contain multiple (on-chip) cache memories and to designs that employ a scratch pad memory (a software-managed on-chip SRAM) instead of conventional caches. It is important to note, however, that if there is a shared level-two (L2) cache in the system, we also need to take into account the energy consumption and performance penalty of its arbitration logic. We assume the existence of a large, off-chip memory (DRAM) whose access time and per access energy consumption are much higher than the corresponding values for on-chip caches. This DRAM space is shared by all processors in the MPoC. We also assume the existence of a simple synchronization mechanism between CPUs to ensure correct execution when they work on the same data in parallel. When a processor is shut-down, we assume that the corresponding caches are also shut-down.

To put a given processor-cache memory pair into the power-down state, different methods can be used [6, 16]. A common characteristic of these techniques is that when the processor/cache is placed into power-down state, the energy consumption is reduced significantly. The magnitude of this energy reduction due to power-down state is expressed as an energy reduction factor (ERF) as will be detailed later in the paper. Another characteristic is that a re-synchronization penalty (RP) is incurred when a processor/cache pair in the power-down state is accessed (attempted to be re-activated). Instead of experimenting with specific values of ERFs and RPs, we chose to vary these parameters to cover a large number of design alternatives. This allows us to evaluate our approach under a variety of scenarios. Note that although we focus here on a single power-down state only, it is also possible to extend our approach to cases where multiple power-down states exist with different ERF and RP values.

This architecture has an important advantage over traditional multi-FU machines such as VLIW/superscalar processors. Since each processor can work independently, we can assign work to each of them at the source code level. For example, we can utilize existing source-level loop parallelization techniques [2, 15] to distribute computation across processors. For array-intensive applications with regular (compile-time predictable) data access patterns, exploiting parallelism opportunities at a large granularity (source level) is expected to return larger benefits than low-level parallelism optimization techniques. In particular, for codes that can be parallelized into multiple threads, the simple MPoC processors working together will perform better than a more complicated wide-issue superscalar machine or a VLIW architecture.

3. PLATFORM AND METHODOLOGY

We used an in-house, cycle-accurate energy simulator [14] to measure the energy consumed in different components of the MPoC such as processors, caches, off-chip memory, clock circuitry, interconnect between processors and caches, and interconnect between caches and off-chip memory. Our simulator takes as input a configuration file and an input code written in C and produces as output the energy distribution across different hardware components and performance data (execution cycles). Each processor is modeled as a simple five-stage pipeline that issues a single instruction at each cycle. Extensive clock gating [5] has been employed to reduce energy consumption.

The energy model used by our simulator for interconnect is transition-sensitive; that is, it captures the switching activity on buses. Since simple analytical energy models for cache memories have proved to be quite reliable, our simulator uses an analytic cache model [12] to capture cache energy. This model takes into account the cache topology, number of hits/misses and write policy, and returns the total amount of energy expended in the cache. We also assume the existence of an off-chip memory and assume a fix per access energy consumption of 4.95 nJ (as in [12]). The simulator uses predefined, transition-sensitive models for each functional unit to estimate the energy consumption of the core. The current implementation does not model the control circuitry in the core. This is not a major problem in this study as the energy consumed by the datapath is much larger than the energy consumed by the control logic due to simple control logic of each MPoC processor (single issue, no speculation). All functional unit energy models are for 0.35 micron technology and have been validated to be accurate (within 10%) [14]. The clock subsystem of the target architecture is implemented using a first level H-tree and a distributed driver approach that supplies clocking to four main units: data cache, instruction cache, register file, and datapath (pipeline registers). The simulated architecture uses static CMOS gates and single-phase clocking for all sequential logic while all memory structures are based on the classic 6T cell. We also model the impact of gating at different levels and for different units. The clock network model was validated to be within 10% error from circuit simulation values. More details can be found in [14].

We also model the energy consumption due to spawning multiple threads to be executed on processors, synchronizing them at the end of each parallel loop, and re-synchronization penalty (waking up time). This is achieved by calculating the energy consumed due to executing the code fragments that perform these spawning/synchronization/re-synchronization tasks. We assume a simple bus-based synchronization mechanism where each processor participates in the synchronization process by setting a bit in its control register. Future work will consider more sophisticated network architectures and more efficient synchronization mechanisms. Unless stated otherwise, all caches in the MPoC are 4KB, 2-way set-associative with a write-back policy and a line (block) size of 32 bytes. The MPoC in consideration has 8 identical processors. The cache access latency is 2 cycles and an off-chip memory access takes 80 cycles.

Figure 2 lists the thirteen benchmark codes used in this study and their important characteristics. `3step-log`, `full- search`, `hier`, and `parallel-hier` are four different motion estimation implementations; `aps`, `bmcm`, `eflux`, and `tsf` are from Perfect Club benchmarks; and `btrix` and `tomcat` are from Spec benchmarks. `adi` is from Livermore kernels and the remaining codes are array-based versions of the DSPstone benchmarks. The second column gives total input size and the third column shows the number of nests in each code. The remaining columns will be explained in the next section.

For each benchmark code, we run three versions: (i) The original version. In this version, we assume that only the processors (and their caches and interconnects) that participate computation consume dynamic energy; but, each processor/cache pair (whether it participates the computation or not) consumes leakage energy. (ii) In this version, which is detailed in Section 4, the processors that do not participate computation are placed into power-down state. Note that, due to the re-synchronization cost, such an approach leads to a performance penalty. (iii) To eliminate this performance penalty, this version (detailed in Section 5) employs a pre-activation strategy using which the processors are pre-activated before they are actually needed. All three versions use clock gating [5] where possible. All necessary code modifications have been implemented using the SUIF compiler infrastructure [1] as a source-to-source translator.

In the rest of this paper, unless otherwise stated, when we mention *energy consumption*, we mean the energies consumed in caches, interconnects, off-chip memory, processor, and the clock circuitry. All optimized energy consumptions are given as values *normalized* with respect to the energy consumed in these components by the original version (version (i)). We do not consider the energy consumed in the control circuitry and I/O modules as the energy impact of our adaptive parallelization strategy on these units is expected to be minimal. It is also possible to extend our power management strategy to selectively shut down these components when the compiler detects that they will not be exercised by a given loop.

4. ADAPTIVE PARALLELIZATION AND IMPACT OF PROCESSOR SHUT-DOWN

Most published work on parallelism [2, 15] is static; that is, the number of processors that execute the code is fixed throughout the execution. In adaptive parallelization, on the other hand, the number of processors is tailored to the specific needs of each code section (e.g., a nested loop in array-intensive applications). For instance, an adaptive parallelization strategy can use 4, 6, 2, and 8 processors to execute the first four nests in a given code. There are two important issues that need to be addressed in designing an effective adaptive parallelization strategy:

•*Mechanism:* How is the number of processors to execute each code section determined? There are two ways of determining the number of processors per nest: dynamic approach and static approach. The first option is adopting a fully-dynamic strategy whereby the number of processors (for each nest) is decided in the course of execution (at run-time). While this approach is expected to generate better results once the number of processors has been decided (as it can take run-time code behavior and resource constraints into account), it may also incur a significant performance overhead during the process of determining the number of processors. This overhead can easily offset the potential benefits of adaptive parallelism. In the second option, the number of processors for each nest is decided at compile-time. This approach has a compile-time overhead but it does not lead to much run-time penalty. In this paper, we adopt the static approach. It should be emphasized, however, that although our approach determines the number of processors statically at compile-time, the activation/deactivation of processors and their caches occurs dynamically at run-time.

•*Policy:* What is the criterion based on which we decide the number of processors? In this study, we used the execution time as the main criterion to decide the number of processors for each nest. This is because our objective is to optimize energy consumption with as little negative impact as possible on performance. In order to determine the number of processors that results in the best execution time for a given nest, we used profile data. That is, using our simulator, we executed the nest with different number of processors and selected the one with the minimum execution time. While this profile-based strategy can increase the compilation time, in many embedded systems, large compilation times can be tolerated as these systems typically run a single (or a small set of) application(s). We plan to make experiments with other criteria (policies) in the future.

It should be mentioned that there are several circumstances that might lead to an increase in overall execution time when the number of processors is increased. As mentioned earlier in the paper, the intrinsic data dependences might lead to excessive synchronization overhead if more processors (than necessary) are used. Also, sometimes (particularly in small loops), the overhead of creating a large number of threads to execute a loop nest in parallel and then synchronizing them after the nest execution can offset the benefits from adaptive parallelism.

Benchmark	N1	N2	N3	N4	N5	N6	N7	N8	N9
`3step-log`	1	1	5						
`adi`	4	5							
`aps`	1	1	1						
`bmcm`	1	1	2	4					
`btrix`	2	1	7	6	1	3	8		
`eflux`	2	3							
`full-search`	2	2	6						
`hier`	1	1	3	3	2	1	5		
`lms`	2	1	2	2					
`n-real-updates`	4	4	4						
`parallel-hier`	3	3	1	1	2				
`tomcat`	2	1	3	1	2	4	1	8	2
`tsf`	1	7	2	4					

Figure 3: Number of processors that generates the best execution time for each nest of each benchmark code in our experimental suite.

Figure 3 shows, for each nest of each benchmark code in our experimental suite, the number of processors that generated the best execution time. The data presented in this figure clearly shows that in many cases, using only a small subset of processors (recall that our MPoC has 8 processors) generates the best result. This is a strong motivation for shutting off unused processors to save energy. The fourth column of Figure 2 gives the average number of

Benchmark Name	Input Size	Number of Nests	Average Number of Processors	Dynamic Energy	Leakage Energy L=0.1	Leakage Energy L=0.5	Leakage Energy L=1
3step-log	203.54KB	3	2.33	85168.83	13046.13	65230.66	130461.31
adi	60.65KB	2	4.50	1695.33	228.22	1141.08	2282.16
aps	137.23KB	3	1.00	680.27	322.63	1613.15	3226.30
bmcm	32.84KB	4	2.00	2033.76	350.46	1752.30	3504.61
btrix	5.89MB	7	4.00	54039.68	6321.93	31609.65	63219.30
eflux	86.70KB	2	2.50	7964.98	1055.58	5277.89	10555.78
full-search	203.54KB	3	3.33	485396.25	59391.22	296956.09	593912.19
hier	203.54KB	7	2.29	47842.54	8306.58	41532.89	83065.78
lms	80.00KB	4	1.75	2419.83	608.04	3040.19	6080.37
n-real-updates	20.00KB	3	4.00	1484.88	177.74	888.72	1777.44
parallel-hier	203.54KB	5	2.00	77027.02	24578.78	122893.91	245787.83
tomcat	70.40KB	9	2.66	9136.33	1306.42	6532.08	13064.17
tsf	52.02KB	4	3.50	2941.12	719.35	3596.73	7193.47

Figure 2: Important characteristics of the benchmarks codes used in our experiments. All energy values are in microJoules.

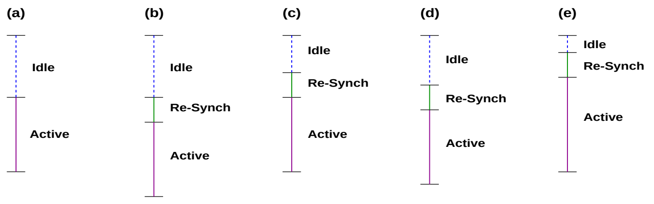

Figure 4: (a) Original idle/active pattern (execution profile). (b) Saving energy using shut-down and incurring performance penalty. (c) Ideal pre-activation strategy. (d-e) Wrong pre-activation timings.

processors per nest for each benchmark.

Having explained our mechanism and policy for determining the number of processors for executing each nest, we next focus on our modifications to the input code. Once the number of processors for each nest has been determined, our strategy inserts (using SUIF) the processor activation/deactivation calls in the code. A processor activation call brings a processor from the power-down state to the active state and takes a specific amount of time to complete (re-synchronization penalty). A deactivation call, on the other hand, places an active processor into the power-down state. We assume that it returns immediately; i.e., it does not incur any additional penalty. Our framework however is general enough to accommodate scenarios where deactivation calls can also have energy and performance costs. Each activation/deactivation call takes the processor id as input parameter and returns a status code indicating whether the action has successfully been performed.

After inserting activation/deactivation calls, our approach performs two optimizations. First, it ensures that in moving from one nest to another, the current active/idle status of processors is maintained as much as possible. The second optimization that we perform targets at reducing the number of activation/deactivation calls inserted in the code when there exists conditional flow of execution. For example, if there is an if-then-else construct with a separate loop in each branch, the compiler hoists the activation/deactivation calls (to be inserted for each loop) above the said construct if both the nests demand the same number of processors.

Note that this energy optimization scheme, while simple to implement, does not pay any attention to reducing the performance overhead due to re-synchronization penalty. Consequently, this approach can result in an increase in execution time. Whether such an increase can be tolerated or not depends largely on the potential energy gains. To illustrate this, let us consider the execution profile of a single processor given in Figure 4(a). The execution profile is broken up into two pieces, each corresponding to a separate loop nest. The processor is idle in the first nest and active (used) in the second. Using our energy-optimization approach gives the modified execution profile shown in Figure 4(b). It is easy to observe from this profile that when the processor is requested to perform computation in the second nest, it first needs to wait some amount of time (re-synchronization penalty, RP). Let t_a, t_i, and t_s denote the active period, idle period, and re-synchronization time, respectively (in cycles). Also, let e_a, e_i, and e_s denote the per cycle energy consumptions for, respectively, the active period, idle period (only when the processor is shut down), and re-synchronization period. Note that, as a result of energy optimization, the length of the original execution profile increases from $t_i + t_a$ to $t_i + t_s + t_a$. The energy consumption of the profile, on the other hand, changes from $(t_i + t_a)e_a$ to $t_i e_i + t_s e_s + t_a e_a$. Assuming conservatively that $e_s = (e_a + e_i)/2$ and $e_i = k e_a$ where k is a coefficient less than one (that is, the energy reduction factor, ERF), this scheme saves energy if and only if:

$$\begin{aligned}(t_i + t_a)e_a &> t_i e_i + t_s(\frac{e_a + e_i}{2}) + t_a e_a \\ &> (k t_i + t_a + \frac{k+1}{2} t_s)e_a.\end{aligned}$$

Consequently, energy savings are possible if:

$$\frac{k+1}{2} t_s < (1-k)t_i.$$

We have evaluated the impact of this energy saving strategy using our benchmark codes. Figure 5 shows the normalized energy consumptions (with respect to version (i)). To cover different scenarios, we performed experiments with different L = (*leakage energy per cycle*)/(*dynamic energy per access*) ratios. This ratio is used for all hardware components of interest. Specifically, we experimented with three different values of L: 0.1, 0.5, and 1. We believe that as leakage is becoming an important part of overall energy budget [6], the experiments with large L values will be more indicative of future trends. In particular, the experiments with L=1 try to capture the anticipated importance of leakage energy in future. Note that leakage is expected to become the dominant part of energy consumption for 0.10 micron (and below) technologies for the typical internal junction temperatures in a chip [6].

The fifth column in Figure 2 gives the overall dynamic energy consumption for version (i). The last three columns show the leakage energy consumption for these three different L values for the same version. In our experiments, we also modified the energy reduction factor (ERF). Specifically, we used ERF=0.1 and ERF=0.2. Note that these values are in reasonable range for several leakage saving techniques such as input vector control and supply gating [6]. The results in Figure 5 include all energy overheads of re-synchronization as well as the energy consumed during spawning threads and synchronizing them following the loop execution. We see that savings for configurations (L=0.1;ERF=0.1) and (L=0.1; ERF=0.2) are 13.9% and 13.3%, respectively. When we increase L to 0.5, the savings for the cases ERF=0.1 and ERF=0.5 move to 43.9% and 41.8%, respectively. Finally, with (L=1.0;ERF=0.1) and (L=1.0;ERF=0.2), the energy savings climb up to 59.7% and

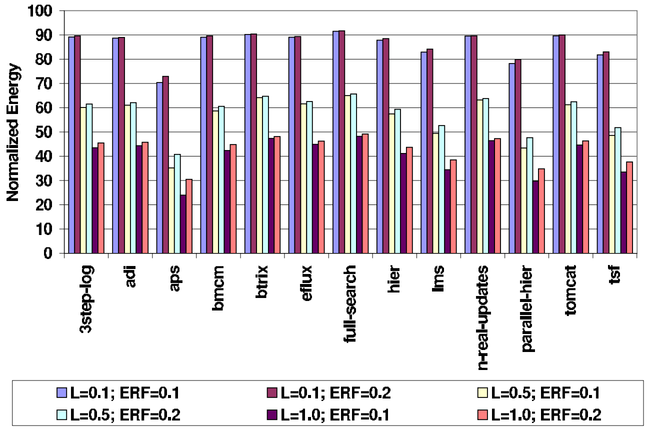

Figure 5: Normalized energy consumption (version (ii)).

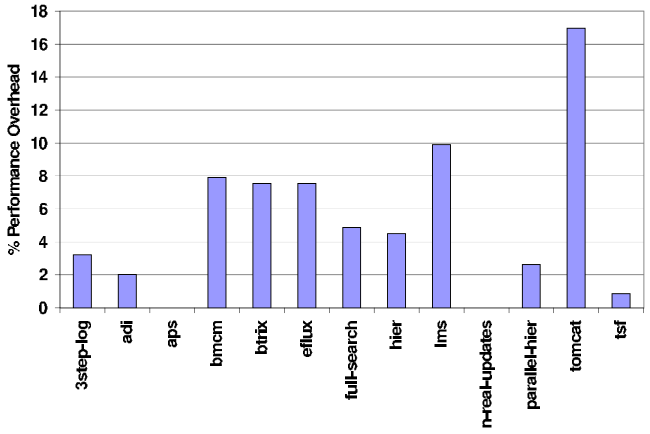

Figure 6: Percentage performance penalty (version (ii)).

57.1%. All these values are obtained assuming that the energy consumed during re-synchronization is the average of active and idle energies and that it takes 20 milliseconds to wake up the processor/cache. Note that this value is very large even for aggressive resource shut-down strategies such as power supply gating [6]. So, in practice, the energy savings that can be obtained from our strategy might be even larger.

While these results indicate large energy savings, delaying waking up a processor until it is actually needed can hurt performance. Figure 6 shows for each benchmark the increase in execution time (over version (i)) when processor shut-down is employed. We observe that the increase in execution time ranges from 0% to 17%, averaging on 5.3%. The reason that two benchmarks (`aps` and `n-real-updates`) do not experience any performance overhead is the fact that each loop in these two codes work with the same number of processors (see Figure 3); consequently, there is no need for processor activation/de-activation between the nests. While this performance penalty may be tolerable for some embedded designs, we also noted that when the re-synchronization penalty was doubled (40 msec), the performance penalties also almost doubled (the detailed results are omitted).

5. PROCESSOR PRE-ACTIVATION

Pre-activation is a strategy that minimizes the performance impact of energy optimizations. It is implemented by activating a resource earlier than the time it will actually be required. The objective here is to eliminate the resynchronization latency that will occur before the resource can start to function normally. Previous work focused on pre-activation of DRAM memory modules [8] and I/O peripherals [3]. In this section, we demonstrate how pre-activation of inactive processors can improve performance and energy behavior of MPoC.

While our processor shut-down strategy explained above can lead to large energy savings, it also increases execution time. For example, employing our approach increases the length of the execution profile shown in Figure 4(a) by t_s cycles (see Figure 4(b)). In this section, we propose a *pre-activation strategy* in which a processor in the power-down state is activated before it is actually needed. The objective here is to ensure that the processor will be ready (i.e., got synchronized) when it is required to participate computation.

The ideal use of this approach is illustrated in Figure 4(c). In this case, the processor remains in the power-down state only for a period of $t_i - t_s$. The last t_s portion of the original idle period is spent in re-synchronization. Consequently, when the processor is requested, it would have just finished the re-synchronization period. An important issue here is to determine the exact point in the code to start re-synchronization. This may not be very trivial; because re-synchronization penalty is given in terms of cycles and it needs to be re-expressed in terms of loop iterations as this (i.e., loop iteration) is the only unit we can use (at source level) to insert activation/deactivation calls. Essentially, in order to pre-activate a processor in the power-down state (for a specific nest), we need to determine an iteration (in the previous nest) before which the processor is activated. Let us assume that each iteration of this previous nest takes C cycles and that the re-synchronization penalty is t_s cycles. Consequently, in the ideal scenario, the processor in question should be activated (i.e., should enter to the resynchronization period) before the last $\lceil \frac{t_s}{C} \rceil$ iterations of the loop. If this is done properly, for our current example, we obtain the execution profile shown in Figure 4(c).

Note that the length of the execution profile in Figure 4(c) is the same as that of Figure 4(a). Its energy consumption, on the other hand, is

$$\begin{aligned} E &= (t_i - t_s)e_i + t_s e_s + t_a e_a \\ &= (kt_i + t_a + \frac{1-k}{2}t_s)e_a \end{aligned}$$

assuming, as before, that $e_s = (e_a + e_i)/2$ and $e_i = ke_a$. Comparing this expression with the original (unoptimized) energy consumption, we can see that this approach is beneficial if

$$\frac{t_s}{2} < t_i.$$

It should be noticed that this ideal pre-activation strategy is better (easier to satisfy) than the power-down scheme discussed in the previous section from both the energy and performance viewpoints.

Figures 4(d) and (e) illustrate, on the other hand, the scenarios where this ideal pre-activation strategy does not happen. In Figure 4(d), the processor activation is delayed. In this case, the length of the original profile is increased (by the amount of the delay in processor activation). In comparison, in Figure 4(e), the processor is activated earlier than necessary. In this case, there is no increase in execution profile length; however, depending on how early the processor is activated, this can cause a significant amount of energy consumption. This is because the re-synchronization period has a fixed length, beyond which the processor is up and starts to consume energy.

The preceding discussion indicates that processor pre-activation can be an effective technique provided that it is done at appropriate moment. There might be several reasons why it may not be possible to achieve ideal pre-activation. First, the point at which the activation call is to be inserted may not be evident from the text of the program. For example, the loop during which a processor needs to be activated (as it is required to participate a computation in the next loop) may have N iterations and our pre-activation strategy can determine that the processor(s) should be activated before the last M iterations are entered. In order to achieve this, we need to split the iteration space of the loop (this is called loop splitting [15]). This, in turn, can increase the code size which may not be tolerated in some embedded environments. Second, the previous

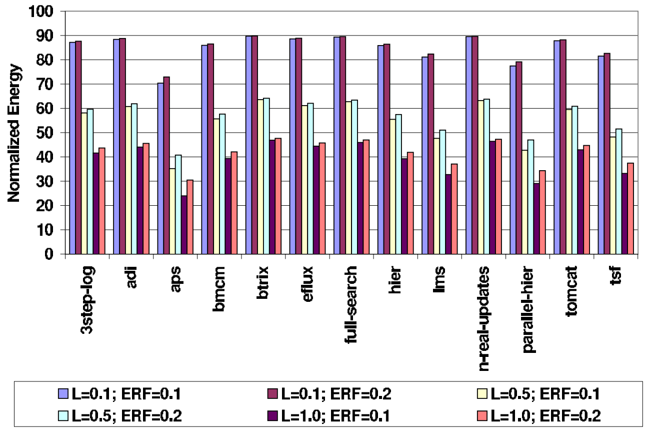

Figure 7: Normalized energy consumption (version (iii)).

loop may not have sufficient number of iterations, in which case we can either consider the next previous nest, or (if it is not possible to do so), we might have to activate the processor(s) later than optimal point, thereby incurring performance penalty.

To evaluate our processor pre-activation strategy (that is, version (iii)), we performed another set of experiments. The results given in Figure 7 (normalized with respect to version (i)) indicate that energy savings due to configurations (L=0.1; ERF=0.1), (L=0.1; ERF=0.2), (L=0.5; ERF=0.1), (L=0.5; ERF=0.2), (L=1.0; ERF=0.1), and (L=1.0; ERF=0.2) are 15.5%, 15.0%, 46.1%, 44.4%, 61.7%, and 59.8%, respectively. The caches (per processor) used in these experiments are 4KB, 2-way set-associative. When we compare these results with the corresponding values given in the previous section, we see that pre-activation is beneficial from an energy perspective too. We also observed that except for one benchmark (`btrix`) the compiler was able to easily insert the pre-activation calls. In `btrix`, in order to insert the pre-activation calls, the compiler needed to split [15] two nests (using SUIF); this led to a 3% increase in the executable size and a 2% degradation in performance. To conclude, processor pre-activation is beneficial from both energy and performance perspectives.

In order to approximate an embedded DRAM, we conducted another group of experiments where the per access off-chip memory energy is reduced to one tenth of its original value. Since this reduces the contribution of DRAM to the overall energy budget (and we do not perform any energy optimization for DRAM), we observed an increase in energy savings due to processor shut-down combined with pre-activation. As an example, with this new value of per-access memory energy, the energy savings for (L=1.0;ERF=0.1) and (L=1.0;ERF=0.2) increased to 68.1% and 63.6%, respectively.

6. FUTURE WORK

This work is a first step in our effort for evaluating and optimizing energy consumption of a complete system based on MPoC. Consequently, it can be extended in several ways:

• We plan to model communication between processors more accurately. Given current trends [6], we can expect that interconnect energy will be more and more pressing in the future. Consequently, to reach an accurate system level energy evaluation, it is critical that interconnect energy should be captured accurately. This issue will be even more important as we attempt to characterize different on-chip interconnection networks from an energy perspective.

•Another way of utilizing multiple processors in an MPoC is to execute (parallelize) code portions *speculatively* when it is not possible to parallelize the application statically. Since speculative parallelization may cause significant wasted energy (in cases where speculation fails), there are important tradeoffs between energy and performance.

•We also plan to augment our current model with energy models of other system components such as I/O devices, software-managed SRAMs, instruction caches, and embedded DRAMs. This will help us to develop more global energy optimization strategies.

•Shutting down idle data caches between nests may not be a good idea when there exists inter-nest data reuse. While this problem did not show up in our examples (as there were not much reuse between nests), a more sophisticated scheme should consider inter-nest reuse before shutting down a data cache. This is an issue that we will visit in the future.

7. CONCLUSIONS

The primary attraction of an MPoC is its ability to shorten execution times for applications that can be parallelized at source level. Due to simple processor architectures (as compared to complex superscalar/VLIW architectures with aggressive branch prediction and speculation) combined with a carefully-designed low-overhead interconnect, these architectures can bring energy benefits as well. Based on the observation that in a given array-intensive code not all the nests require the maximum number of processors in the MPoC, in this paper, we evaluated an adaptive loop parallelization strategy combined with selective shut-down of unused processors. To eliminate potential performance penalty due to energy management, we also proposed a processor pre-activation strategy. Our experiments with an 8-processor MPoC and different parameters (e.g., cache size, re-synchronization overhead, and per-access off-chip memory energy) indicated that our approach is successful in reducing energy consumption.

8. REFERENCES

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng The SUIF compiler for scalable parallel machines. In Proc. *the Seventh SIAM Conference on Parallel Processing for Scientific Computing,* February, 1995.

[2] U. Banerjee. *Loop Parallelization.* Kluwer Academic Publishers, Boston, 1994.

[3] L. Benini, G. De Micheli. System-level power optimization: techniques and tools. *TODAES* 5(2): 115-192 (2000).

[4] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with Piranha. *Technical Report 954,* Yale University, February 1993.

[5] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[6] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[7] Chip Multiprocessing. *ITWorld.Com,* http://www.itworld.com/Comp/1092/CWSTO54343/

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In Proc. *the 7th International Conference on High Performance Computer Architecture,* Monterrey, Mexico, January 2001.

[9] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers,* September 1999.

[10] MAJC-5200. http://www.sun.com/microelectronics/MAJC/5200wp.html

[11] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single chip multiprocessor. In Proc. *the 7th Int'l Conference on Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 2–11.

[12] W.-T. Shiue and C. Chakrabarti. Memory exploration for low power embedded systems. In Proc. *the Design Automation Conference,* 1999.

[13] I. Verbauwhede and C. Nicol. Low power DSPs for wireless communications. In Proc. *ISLPED'00,* Rapallo, Italy, 2000.

[14] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In Proc. *International Symposium on Computer Architecture*, June 2000.

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

[16] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y. Tsai. Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In Proc. *the 34th Annual International Symposium on Microarchitecture,* Austin, TX, December 2001.