Optimizing Array-Intensive Applications for On-Chip Multiprocessors

Ismail Kadayif, *Student Member*, *IEEE*, Mahmut Kandemir, *Member*, *IEEE*, Guilin Chen, *Student Member*, *IEEE*, Ozcan Ozturk, *Student Member*, *IEEE*, Mustafa Karakoy, and Ugur Sezer, *Member*, *IEEE*

Abstract—With energy consumption becoming one of the first-class optimization parameters in computer system design, compilation techniques that consider performance and energy simultaneously are expected to play a central role. In particular, compiling a given application code under performance and energy constraints is becoming an important problem. In this paper, we focus on an on-chip multiprocessor architecture and present a set of code optimization strategies. We first evaluate an adaptive loop parallelization strategy (i.e., a strategy that allows each loop nest to execute using a different number of processors if doing so is beneficial) and measure the potential energy savings when unused processors during execution of a nested loop 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 percent energy savings at the expense of up to 17 percent performance loss in a set of array-intensive applications. To eliminate this performance penalty, we also discuss and evaluate a processor preactivation 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 preactivation can be very effective in reducing energy consumption without increasing execution time. We then generalize our strategy and present an application parallelization strategy based on integer linear programming (ILP). Given an array-intensive application, our optimization strategy determines the number of processors to be used in executing each loop nest based on the objective function and additional compilation constraints provided by the user/programmer. Our initial experience with this constraint-based optimization strategy shows that it is very successful in optimizing array-intensive applications on on-chip multiprocessors under multiple energy and performance constraints.

Index Terms—On-chip multiprocessor, constrained optimization, embedded systems, energy consumption, adaptive loop parallelization, integer linear programming.

1 Introduction

As the applications ported into System-on-a-Chip (SoC) architectures become more and more complex, it is extremely important to have sufficient compute power on the chip. One way of achieving this is to put multiple processor cores in a single chip. This "on-chip multiprocessing" strategy has several advantages over an alternate strategy, which puts a more powerful and complex processor in the chip. First, the design of an on-chip multiprocessor composed of multiple simple processor cores is generally simpler than that of a complex single processor system [27], [40]. This simplicity also helps reduce the time spent in verification and validation [38]. Second, an on-chip multiprocessor is expected to result in better utilization of the silicon space. The extra logic that would be spent on register renaming, instruction wake-up, speculation/prediction, and register bypass on a complex single processor can be spent for providing higher bandwidth on an on-chip multiprocessor. Third, an on-chip multiprocessor architecture can exploit loop-level parallelism at the software level in array-intensive applications. In contrast, a complex single processor architecture needs to convert loop-level parallelism to instruction-level parallelism at runtime (that is, dynamically) using sophisticated (and power-hungry) strategies. During this process, some loss in parallelism is inevitable. Overall, an on-chip multiprocessor is a suitable platform for executing array-intensive computations commonly found in embedded image and video processing applications (though it may not be the ideal platform for control-intensive applications).

While VLIW/superscalar processors may provide a certain degree of (instruction level) parallelism, as noted by Verbauwhede and Nicol [49], 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 (though VLIWs are better than superscalar machines on these aspects). Recently, automatic loop parallelization technology developed for array-intensive applications has been shown to be very effective [51]. We believe that array-intensive embedded applications can also take advantage of this technology and derive significant performance benefits from

- *I. Kadayif is with the Department of Computer Engineering, Canakkale Onsekiz Mart University, 17100 Canakkale, Turkey. E-mail: kadayif@comu.edu.tr.*
- *M. Kandemir, G. Chen, and O. Ozturk are with the CSE Department, The Pennsylvania State University, University Park, PA 16802. E-mail: {kandemir, guilchen, ozturk}@cse.psu.edu.*
- *M. Karakoy is with the Department of Computing, Imperial College, London SW7 2BZ, UK. E-mail: mk22@doc.ic.ac.uk.*
- *U. Sezer is with the ECE Department, University of Wisconsin, Madison, WI 53706. E-mail: sezer@ece.wisc.edu.*

Manuscript received 15 Oct. 2003; revised 9 Aug. 2004; accepted 31 Aug. 2004; published online 22 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0190-1003.

the on-chip parallelism and low-latency synchronization provided by an on-chip multiprocessor.

An on-chip multiprocessor improves execution time of applications using on-chip parallelism. An application program can be made to run faster by distributing the work it does over multiple processors on the on-chip multiprocessor. There is always some part of the program's logic that has to be executed serially, by a single processor; however, in many applications, it is possible to parallelize some parts of the program code. Suppose, for example, that there is one loop in the code where the program spends 50 percent of its execution time. If the iterations of this loop can be divided across two processors so that half of them are done in one processor while the other half are done at the same time in the other processor, the whole loop can be finished in half the time, resulting in a 25 percent reduction in overall execution time. While this argument favors increasing the number of processors as much as possible, there is a limit beyond which using a larger number of processors might actually degrade performance. This is because, in many cases, parallelizing a loop entails interprocessor communication/synchronization. Increasing the number of processors in general increases the frequency and volume of this activity. Therefore, after a specific number of processors is reached, increasing the number of processors further increases communication/synchronization costs so significantly that additional benefits due to more parallelism may not be able to offset this (note that, in a sense, this is a consequence of the Amdahl's Law).

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 interprocessor communication/synchronization costs. Similarly, a small loop bound may also suggest the use of fewer processors (than available) to execute a given loop. Loops, in particular, present an excellent optimization scope for adaptive parallelization. Since, in general, each loop in a given application might require a different number of processors to achieve its best performance, it might be useful to change the number of processors across the loops. Previous research on large scale high-end parallel machines [42], [9], [41] 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 the application) can be effective in maximizing the utilization of processors.

When adaptive loop parallelization is employed in a given on-chip multiprocessor, the unused (idle) processors can be shut down to conserve energy. Depending on the inherent degree of parallelism in different loop 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 "resynchronization penalty" when a processor placed into a power-down (sleep) state is requested to participate in computation (e.g., in the next loop 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 "preactivate" a processor before it is actually needed so as to ensure that it is ready when it is required to perform the next computation. Such a preactivation strategy can, if successful, eliminate the performance penalty due to resynchronization and reduce energy consumption.

Even if one focuses only on performance, selecting the number of processors to use in parallelizing loop nests may not be trivial. When we consider multiple objective functions at the same time, the problem becomes even more difficult to address. Suppose, for example, that we would like to minimize the energy-delay product of a given loop using parallelization. In order to decide the number of processors to use, we need to evaluate the impact of increasing the number of processors on both energy consumption and execution cycles. If using more processors does not bring significant reductions in execution time, the energy-delay product may suffer, as using more processors means that more processors should be powered on. Evaluating the impact of increasing the number of processors on both energy and performance is difficult if one restricts itself only to static (compile-time available) information. Finally, the possibility that each loop in a given application may demand a different number of processors to generate the best results makes the overall on-chip code parallelization problem under multiple constraints highly complex.

In this paper, we focus on a constraint-based optimization problem for on-chip multiprocessors and make the following contributions:

- We evaluate an adaptive code parallelization strategy (which makes use of profiling) and measure the potential energy savings when unused processors in an on-chip multiprocessor are shut down. Our results show that shutting down unused processors can lead to as much as 67 percent energy savings at the expense of up to 17 percent performance loss in a set of array-intensive applications.
- We present a processor preactivation 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 preactivation can be very effective in reducing energy consumption without increasing execution time. Our experiments with preactivation indicate a 39 percent reduction in energy consumption (on average) as compared to a scheme without energy management.
- We propose an integer linear programming (ILP) based strategy to determine the ideal number of processors to use in parallelizing each loop nest in an application code to be run on an on-chip multiprocessor. Our strategy has two components. First, there is a "profiling phase," where we run (simulate) each loop nest with a different number of processors and collect performance and energy data. In the second phase, called the "optimization phase," we formulate the problem of selecting a processor size (i.e., the number of processors) for each parallel loop as an integer linear programming (ILP) problem and solve it using a publicly-available tool. Our ILP-based approach takes into account multiple constraints that can involve execution time and energy consumption on different hardware components. To the best of our knowledge, this is the first multi-constraint-based application parallelization study for on-chip multiprocessors. We implemented our

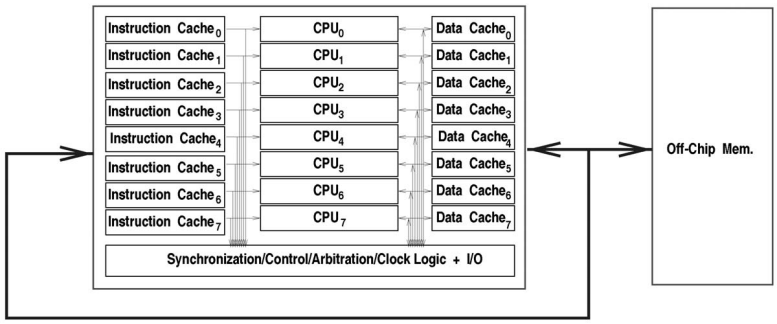

Fig. 1. On-chip multiprocessor architecture and off-chip memory. This is a shared-memory architecture, that is, the entire memory space is shared by all processors.

ILP-based strategy using an experimental compiler and customized simulator environment and measured its effectiveness by applying it to a number of array-intensive applications. Our results indicate that this strategy is very successful in determining the ideal number of processors (for each loop nest in an array-based application), even under multiple complex constraints. Our experience also shows that the ILP-based approach is fast in practice. More specifically, the maximum time spent in solving the ILP problem for all applications tested was always below 100 milliseconds (on a 400MHz SunSparc-based machine). Since optimizers for embedded systems can afford longer processing times as compared to their general-purpose counterparts, we believe that this optimization time is tolerable.

The remainder of this paper is organized as follows: Section 2 gives an overview of the on-chip multiprocessor architecture considered in this paper. Section 3 discusses our experimental platform. Section 4 demonstrates why parallelization for minimum execution time and parallelization for minimum energy consumption can demand different strategies. Section 5 discusses how adaptive parallelization can be employed for improving energy consumption without degrading performance. Section 6 presents the details of our constraint-based parallelization strategy and gives experimental results. Section 7 concludes the paper by summarizing our major contributions.

2 On-Chip Multiprocessor and Loop Parallelization

In this paper, we focus on an on-chip multiprocessor architecture. The abstract view of this parallel architecture is shown in Fig. 1. This architecture contains multiple processors (each with its own instruction and data caches) and an interprocessor synchronization and clock logic. This is a shared memory architecture; that is, all interprocessor communication occurs through reading from and writing into a shared off-chip memory (also shown in the figure). A bus-based on-chip interconnect is used to perform interprocessor synchronization. Such synchronization is necessary for the processors to be synchronized at the beginning and end of each loop nest they execute. Although we could include an L2 cache to this architecture (as in the case of [40]), in this work, we have not done so as the existence of L2 would not change the main observations made in this paper.

As mentioned in the previous section, an on-chip multiprocessor architecture has some advantages compared to out-of-order superscalar machines. Maybe the most important of these is the simplicity and independence of the processors. The processors we assume in this study are single-issue, five-stage pipelined architectures without any complex branch prediction or data speculation logic. This brings an important side-advantage in terms of execution time predictability as it is easier to predict execution time with simple processors without sophisticated prediction/speculation logic (a big plus in real-time embedded environments). Also, each processor can operate independently from each other, and the processors engage in synchronization only to maintain data integrity during parallel execution.

On-chip multiprocessors are very suitable for array-intensive embedded applications. This is because many array-intensive embedded applications are composed of multiple independent loop nests that operate on large, multidimensional arrays of signals. An important issue in such applications is to map the application code to the embedded architecture in question. Note that, in the on-chip multiprocessor case, this is relatively simple as, in the source-code level, we can parallelize each loop nest and distribute loop iterations across the processors in the on-chip multiprocessor. If the loop in question is parallelizable, high execution time benefits can be achieved using this strategy. In contrast, a superscalar or a VLIW architecture should map parallel loop iterations to parallel (and sometimes irregular) functional units, which may not form a good match for the application structure.

An array-intensive embedded application is executed on our on-chip multiprocessor architecture by "parallelizing its loops." Specifically, each loop is parallelized such that its iterations are distributed across processors. An effective parallelization strategy should minimize the interprocessor 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 large off-chip shared memory can be very costly from both the execution cycles and energy consumption perspectives.

During execution, when a parallel loop is reached, multiple parallel threads are initiated (spawned). When the parallel execution of the loop is completed, the threads (each running on a processor) synchronize. If the next loop

is also parallel, again multiple threads are spawned. Otherwise (if the next loop is sequential), a single thread executes it. Our experimental evaluation takes into account all energy/performance overheads of thread creation, thread termination, and synchronization.

Bahar and Manne [2] present pipeline balancing, a technique that dynamically tunes the resources of a general-purpose processor to the needs of the program. Marculescu [35] presents a profile-driven energy optimization strategy for superscalar architectures. Conte et al. [14] also discuss adaptive execution for power efficiency. Our work is different from these in at least two important aspects. First, we focus on an on-chip multiprocessor architecture instead of a superscalar machine. Consequently, our analysis necessary to derive the opportunities for resource shut down is very different from these studies. Second, we explicitly focus on loop-based applications and use a compiler to shut down processors completely (not just one of their functional units and part of their issue width) based on the degree of loop-level parallelism under different criteria.

Apart from academic interest [40], [27], chip multiprocessor architectures are also finding their way into commercial products. A number of architectures, such as IBM's Power4, Sun's MAJC-5200, and MP98, contain multiple processors (currently between two and four) on a single die [34], [37], [13]. To quote a recent news article [12]: "Sun and IBM agree on one thing: chip multiprocessing is the next big thing in CPU design. In the past, engineers squeezed out faster performance with better process technology, better microarchitectures, and better compilers. Chip multiprocessing adds another performance tool that directly addresses the Holy Grail of processor evolution. This could take us beyond Moore's Law. That is great news for users but one more challenge for software programmers." Therefore, we believe that software support for on-chip multiprocessor will be very important in the future.

More recently, there have been several efforts for obtaining accurate energy behavior for on-chip processing and communication (e.g., see [16] 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 on-chip multiprocessor architectures. Several recent studies have also considered energy consumption for parallel architectures. [28] explores the possibility of using single-ISA heterogeneous cores to attack the power consumption problem. Since the previous generation cores are much smaller and consume less power, dynamically switching between cores can bring better energy efficiency. In [32], Li et al. argue that a core can be put into sleep mode when it reaches the barrier early. An earlier study [43] have pointed out that the CMP can be an energy-efficient alternative to exploiting future billion transistor designs and also mentioned that voltage scaling can further complement this architecture. They show around 9 percent to 15 percent power savings in multimedia applications that use independent threads. [24] considers voltage scaling in a multiprocessor-on-chip type of architecture. In comparison to these studies, this paper considers a compiler-driven approach to reduce power consumption through processor shutdown. It also studies the trade offs between energy, performance, and code size.

3 Experimental Platform

In this section, we discuss our experimental platform and introduce our benchmarks. We used an in-house, cycle-accurate energy simulator [50] to measure the energy consumed in different components of the on-chip multiprocessor, such as processors, caches, off-chip memory, clock circuitry, interconnect between processors and caches, and interconnect between caches and off-chip main 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 (with the traditional IF, ID, EXE, MEM, and WB stages) that issues a single instruction at each cycle. Extensive clock gating [10] has been employed to reduce energy consumption in the pipeline.

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 [25], our simulator uses an analytic cache model [46] to capture cache energy (for 0.1 micron technology). 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 fixed per access energy consumption of 4.95 nJ (as in [46]). The simulator uses predefined, transition-sensitive models for each functional unit to estimate the energy consumption of the core [11]. These transition-sensitive models contain switch capacitances for a functional unit for each input transition obtained from VLSI layouts and extensive HSPICE simulation. Once the functional unit models have been built, they can be reused for many different architectural configurations. The current implementation does not model the control circuitry in the core. This is not a major problem in this study since the energy consumed by the datapath is expected to be much larger than the energy consumed by the control logic due to the simple control logic of our on-chip processors (i.e., single issue, no speculation). All functional unit energy models used have been validated to be accurate (within 10 percent) [50]. 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 percent error from circuit simulation values. More details can be found in [18]. Default parameters used in our simulations are listed in Fig. 2 (some of these parameters are later modified for exploring different aspects of our strategy). Since not all processors are used in executing a given loop nest, the unused processors and their instruction and data caches can be placed into a power-down (sleep) mode (state). In the power-down state, the processor and caches consume only a small percentage of their original (per cycle) leakage energy. However, when a processor and its data and instruction caches in the sleep state are needed, they need to be reactivated (resynchronized). This resynchronization costs extra execution cycles as well as extra energy

Simulation Parameter	Value
Processor Speed	400MHz
Number of Processors	8
Instruction Cache	4KB 2-way associative 32 byte blocks 2 cycle latency
Data Cache	4KB 2-way associative 32 byte blocks 2 cycle latency
Memory	32 MB 80 cycle latency
Cache Dynamic Energy Consumption	0.60 nJ/access
Off-Chip Memory Dynamic Energy Consumption	4.95 nJ/access
Leakage Energy Consumption for 32 bytes Active State Sleep State	 4.49 pJ/cycle 0.92 pJ/cycle
Resynchronization Time	20 msec

Fig. 2. Default simulation parameters used in this study. Some of these parameters are later modified to conduct a sensitivity analysis.

consumption. In this study, we assumed a resynchronization latency of 20 msec (a very conservative estimate) and a full leakage energy consumption during resynchronization period. Also, we model a state-preserving cache turn-off strategy, similar to that proposed in [19]. In this architecture, when the cache is turned off, the contents are maintained. Since in our on-chip multiprocessor interprocessor synchronization is through a bus-based mechanism, we modeled its energy consumption as that of placing 8 bits into the interconnect (one for each processor) and the associated control register update (to set/reset the synchronization bit). Note that processors share data through off-chip memory; that is, there is no extra interconnect for explicit interprocessor data communication. 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 resynchronization penalty (waking up time). This is achieved by calculating the energy consumed due to executing the code fragments that perform these spawning/synchronization/resynchronization tasks. Unless stated otherwise, all caches in our on-chip multiprocessor are 4KB, 2-way set-associative with a write-back policy and a line (block) size of 32 bytes. The on-chip multiprocessor in consideration has eight identical processors. The cache access latency is 2 cycles and an off-chip memory access takes 80 cycles. In this work, our focus is on the on-chip multiprocessor and we do not apply any energy optimization to the off-chip memory.

An important energy component of interest is the energy spent during cache coherence. In this work, we assume a MESI-based coherence scheme across the caches. MESI is an invalidation-based protocol for write-back caches used in multiprocessor machines. In this protocol, a cache line can be in one of four states: modified (M) or dirty, exclusive-clean (E), shared (S), and invalid (I); and each cache maintains the state information for all the lines it currently has. The state I means that the line is invalid. M indicates that only the cache under consideration has a valid copy of the line, and the copy in the main memory is stale. E means that only the cache under consideration has a copy of the line which is the same as the corresponding copy in the main memory. Finally, S means that potentially two or more processors have this line in their caches in an unmodified form. Note that state E helps reduce bus traffic for sequential portions of the code where data is not shared. More detailed information on MESI protocol can be found elsewhere [15]. It should be noted that MESI is a snoop-based protocol, which means that processors continuously observe the traffic on the bus to take appropriate coherence actions. Therefore, as far as energy consumption is concerned, there are two major cost components: snooping the bus for every action consumes energy and executing the protocol itself consumes energy. An important characteristic of our simulator is that it can model a MESI-based system (including the energy spent on bus snooping, cache lookups, maintaining state transitions, and cache line invalidations). Therefore, all the energy results that we present include the energy cost of the coherence activity as well.

Fig. 3 lists the 13 array-based 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 [6]; and `btrix` and `tomcatv` are from Spec benchmarks. `adi` is from Livermore kernels and the remaining codes are array-based versions of the DSPstone benchmarks [54]. The second column gives total input size and the third column shows the number of loop nests in each code. The remaining columns of this table will be discussed later.

All necessary code modifications (including the preactivation strategy that will be discussed later in the paper) have been implemented using the SUIF compiler infrastructure [1]

Benchmark Name	Input Size	Number of Loop Nests	Average Number of Processors	Dynamic Energy	Leakage Energy		
					L=0.1	L=0.5	L=1
3step-log	203.54KB	3	2.33	85168.831	13046.127	65230.661	130461.310
adi	60.65KB	2	4.50	1695.330	228.224	1141.080	2282.162
aps	137.23KB	3	1.00	680.268	322.631	1613.146	3226.302
bmcm	32.84KB	4	2.00	2033.761	350.461	1752.298	3504.608
btrix	5.89MB	7	4.00	54039.682	6321.926	31609.647	63219.303
eflux	86.70KB	2	2.50	7964.981	1055.575	5277.888	10555.778
full-search	203.54KB	3	3.33	485396.247	59391.218	296956.089	593912.187
hier	203.54KB	7	2.29	47842.538	8306.580	41532.890	83065.776
lms	80.00KB	4	1.75	2419.827	608.044	3040.192	6080.374
n-real-updates	20.00KB	3	4.00	1484.882	177.742	888.720	1777.437
parallel-hier	203.54KB	5	2.00	77027.023	24578.782	122893.913	245787.832
tomcatv	70.40KB	9	2.66	9136.330	1306.421	6532.083	13064.166
tsf	52.02KB	4	3.50	2941.119	719.353	3596.727	7193.474

Fig. 3. Important characteristics of the benchmark codes used in our experiments. All energy values are in microjoules. L is the ratio between the "leakage energy consumption per cycle" and the "dynamic energy consumption per access."

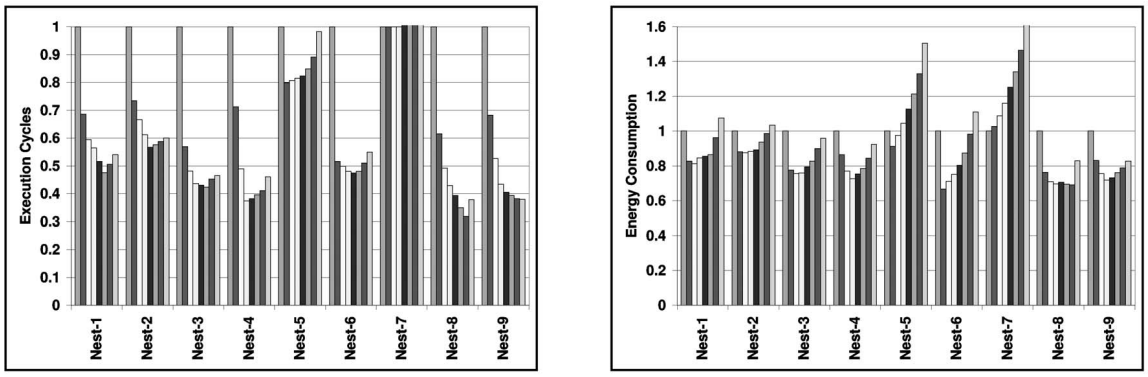

Fig. 4. (a) Normalized execution cycles and (b) energy consumption with different processor sizes (`tomcatv`). Note that energy and performance trends are different from each other.

as a source-to-source translator. SUIF consists of a small kernel and a toolkit of compiler passes built on top of the kernel. The kernel defines an intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. The toolkit includes a C front end, a loop-level parallelism and locality optimizer, an optimizing back end, and a set of compiler development tools.

In the rest of this paper, unless otherwise stated, when we mention "energy consumption," we mean the sum of the energy consumptions in caches, interconnects, off-chip memory, processor, and the clock circuitry (all energy overheads including thread management, interprocessor synchronization, and cache coherence). All optimized energy consumptions are given as values "normalized" with respect to the energy consumed in these components by the original (unoptimized) version of the code in question. We do not consider the energy consumed in the control circuitry and I/O modules since the energy impact of our optimizations 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 nest.

4 Impact of On-Chip Parallelization on Energy and Performance

In order to see whether parallelization for energy and parallelization for performance generate the same behavior or not, we conducted an initial set of experiments where we run each loop nest of each code in our experimental suite using different numbers of processors (ranging from 1 to 8) and measured the energy consumption and execution cycles. Since the trends for different benchmarks were similar, we focus here only on `tomcatv`, one of our benchmarks.

Fig. 4 gives the execution time and energy consumption for each of the nine loop nests of this benchmark. For each loop nest, the eight bars from left to right correspond to different numbers of processors from one to eight. Also, each bar represents a result *normalized* with respect to the single processor case (that is, the energy consumption or execution cycles when the loop nest in question is executed on a single processor).

From these graphs, we can make several observations. First, energy and performance trends are different from each other. That is, in a given loop nest, the best processor size from the energy perspective is (in general) different from the best processor size from the execution time perspective. This is because, in many cases, increasing the number of processors beyond a certain point may continue to improve performance, but the extra energy to power on the additional processor (and its caches) may offset any potential energy benefits coming from the reduced execution cycles. Second, for a given nest, increasing the number of processors does not always reduce execution cycles. This occurs because there is an extra performance cost of parallelizing a loop, which includes spawning parallel threads, interprocessor synchronization during loop execution if there are data dependences, and synchronizing the threads after the loop execution. If the loop has not sufficient number of iterations to justify the use of a certain number of processors, this extra cost can dominate the overall performance behavior. Similarly, beyond a processor size, the energy consumption starts to increase. Third, the best number of processors from the energy or performance perspectives depends on the loop nest in question. That is, different loop nests may require different processor sizes for the best results, and it does not seem to be a good idea to use the same number of processors for all the nests in a given application. Given these observations, one can see that it is not easy to determine the ideal number of processors for each loop nest in a given code to optimize an objective function that may have both performance and energy elements. As an example, consider a compilation (parallelization) scenario where we would like to optimize energy consumption of the application keeping the execution time under a predetermined limit. Important questions in this scenario are 1) whether there are processor sizes for each loop nest to satisfy this compilation constraint and 2) if there are multiple solutions, how can one choose the best one? In the following section, we present a constraint-based adaptive loop parallelization strategy that addresses this problem.

5 Optimizing for Energy under Maximum Performance Based on Adaptive Parallelism

5.1 Energy Benefits of Adaptive Parallelization

Most published work on parallelism [3], [51] is based on static techniques, that is, the number of processors that

Benchmark Name	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					

Fig. 5. The number of processors that generate the best execution time for each loop nest of each benchmark code in our experimental suite. One can see that, in general, each loop nest demands a different number of processors for the best execution time.

execute the code is fixed for the entire execution. For example, if the number of processors that execute a code is fixed at eight, all parts of the code (e.g., all loop nests) are executed using the same eight processors. In adaptive parallelization, on the other hand, the number of processors can be tailored to the specific needs of each code section (e.g., a nested loop in array-intensive applications). In other words, the number of processors that are active at a given period of time changes dynamically as the program executes. For instance, an adaptive parallelization strategy can use four, six, two, and eight processors to execute the first four loop nests in a given code. There are two important issues that need to be addressed in designing an effective adaptive parallelization strategy for an on-chip multiprocessor:

- *Mechanism:* How is the number of processors to execute each code section determined? There are at least two ways of determining the number of processors (per loop nest): the dynamic approach and the 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 runtime). While this approach is expected to generate better results once the number of processors has been decided (as it can take runtime code behavior and dynamic resource constraints into account), it may also incur some performance overhead during the process of determining the number of processors. This overhead can, in some cases, offset the potential benefits of adaptive parallelism. In the second option, the number of processors for each loop nest is decided at compile-time. This approach has a compile-time overhead but it does not lead to much runtime penalty. In this paper, we adopt a profile-driven 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 runtime.
- *Policy:* What are the criterion which we decide the number of processors based on? An optimizing compiler can target different objective functions, such as minimizing execution time of the compiled code, reducing executable size, improving power or energy behavior of the generated code, and so on. In this section, we use reducing execution time as the main goal in deciding the number of processors to be used for each loop nest (in the next section, we present a more general optimization strategy). In other words, our objective is to optimize energy consumption with as little negative impact as possible on performance. More specifically, for each loop nest, we try to determine the number of processors that lead to faster execution than the other alternatives. In order to determine the number of processors that result in the best execution time for a given loop nest, we employed profiling. That is, using our simulator, we executed the loop nest with different numbers 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 since these systems typically run a single (or a small set of) application(s) and the code quality is very critical.

Fig. 5 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 illustrates that, in many cases, using only a small subset of processors (recall that our on-chip multiprocessor has a total of eight processors) generates the best performance. Clearly, this is a strong motivation for shutting off unused processors to save energy. The fourth column of Fig. 3 shows the average number of processors (that gives the "best execution time") per loop nest for each benchmark.

Having explained our mechanism and policy for determining the number of processors for executing each loop nest, we next focus on our modifications to the input code. Once the number of processors for each loop nest has been determined, our strategy inserts (using SUIF [1]) 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 (resynchronization 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. It should be emphasized that

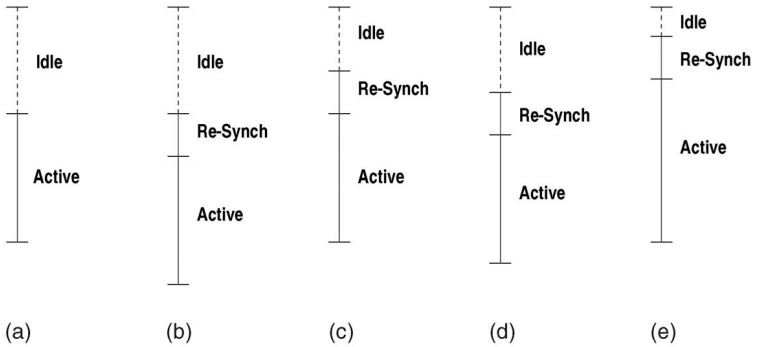

Fig. 6. (a) Original idle/active pattern (execution profile) for a given processor. (b) Saving energy using shut down and incurring performance penalty. (c) Preactivation strategy. (d)-(e) Wrong preactivation timings. Note that the point at which resynchronization activity starts influences both performance and energy consumption.

a wrong placement of activation/deactivation calls does not create a correctness issue, though it may negatively affect the overall performance. Also, the energy values reported in this paper include the energy overheads of the activation/deactivation calls.

After inserting activation/deactivation calls, our approach performs two optimizations. First, it ensures that in moving from one loop nest to another, the current active/idle status of processors is maintained as much as possible. As an example, in moving from a loop nest running with k processors to a loop nest that will be executing using $j\ (> k)$ processors, the same k processors that are used in the first loop nest are used in the second one as well; only an additional $j - k$ processors are activated. Similarly, in moving from a loop nest running with k processors to a loop nest that will be executing using j processors where $j < k$, the only action performed is to deactivate $k - j$ processors of those executing the first nest. The second optimization that we perform targets at reducing the number of activation/deactivation calls inserted in the code when there exists a conditional flow of execution. For example, if there is an if-then-else construct with a separate loop nest in each branch, the compiler hoists the activation/deactivation calls (to be inserted for each loop) above the said construct if both the loop nests demand the same number of processors. A similar optimization is also used at the program points where multiple flows of control merge. The details of this optimization are omitted as their discussion is not essential for the study presented in this paper.

This energy optimization scheme does not pay any attention to reducing the performance overhead due to resynchronization 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 Fig. 6a. The execution profile is broken up into two pieces, each corresponding to a separate loop nest. The processor is idle in the first loop nest and active (used) in the second one. Using our energy-optimization approach gives the modified execution profile shown in Fig. 6b. Note that, here, the processor is "shut down" in the idle period. It is easy to observe from this profile that when the processor is requested to perform computation in the second loop nest, it first needs to wait some amount of time (resynchronization penalty, RP). Let t_a, t_i, and t_s denote the active period, idle period, and resynchronization 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 resynchronization period. 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 $e_s = (e_a + e_i)/2$[1] 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:

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

Consequently, energy saving is 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. For each benchmark code, we have run three versions of it: 1) The original version. In this version, only the processors (and their caches and interconnects) that participate in computation consume dynamic energy, but every processor/cache pair (whether they participate in the computation or not) consume leakage energy. 2) In this version, which is detailed in this section, the processors that do not participate in computation are placed into the power-down state. Due to the resynchronization cost, such an approach leads to a performance penalty. 3) To eliminate this performance penalty, this version (which is detailed in Section 5.2) employs a preactivation strategy, using which the processors/caches are preactivated before they are actually needed. All three versions use clock gating [10] where possible.

Fig. 7 shows the energy consumptions for version 2 ("normalized" with respect to version 1). To cover different scenarios, we performed experiments with different L = (*leakage energy per cycle*)/(*dynamic energy per access*) values. 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 [10], [8], the experiments with large L values (e.g., 1) will be more indicative of future trends. In particular, our experiments with L = 1 aims at capturing the anticipated importance of

1. This assumption is not critical, and our experiments with other values of e_s, including $e_s = e_a$, showed also similar trends.

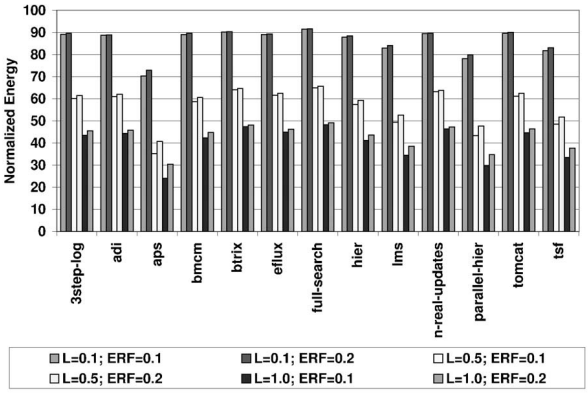

Fig. 7. Normalized energy consumption (version 2). All the values are obtained assuming that a full leakage energy is consumed during the resynchronization periods.

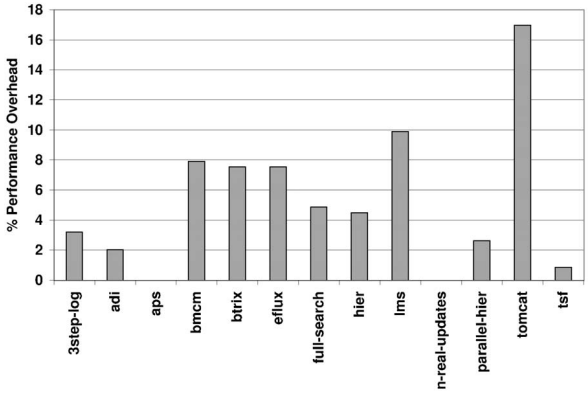

Fig. 8. Percentage performance penalty (version 2). It is assumed that it takes 20 milliseconds to wake up the processor/cache.

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 [10]. All other parameters are as given in Fig. 2.

The fifth column in Fig. 3 gives the overall dynamic energy consumption for version 1. 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 (if not conservative) for several leakage saving techniques such as [19], [26], [31]. *The results in Fig. 7 include all energy overheads associated resynchronization, thread spawning and synchronization, and cache coherence activities.* We see that savings for configurations (L = 0.1; ERF = 0.1) and (L = 0.1; ERF = 0.2) are 13.9 percent and 13.3 percent, respectively. When we increase L to 0.5, the savings for the cases ERF = 0.1 and ERF = 0.2 move to 43.9 percent and 41.8 percent, respectively. Finally, with (L = 1.0; ERF = 0.1) and (L = 1.0; ERF = 0.2), the energy savings climb up to 59.7 percent and 57.1 percent. All the values are obtained assuming that a full leakage energy is consumed during the resynchronization periods and that it takes 20 milliseconds to wake up the processor/cache.

While these results indicate large energy savings, delaying waking up a processor until it is actually needed can hurt performance. Fig. 8 shows, for each benchmark, the increase in execution time (over version 1) when processor shut down is employed (i.e., when version 2 is used). We observe that the increase in execution time ranges from 0 percent to 17 percent, averaging on 5.3 percent. 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 Fig. 5); consequently, there is no need for processor activation/deactivation between the loop nests. While this performance penalty may be tolerable for some embedded designs, we also noted that when the resynchronization penalty was doubled (40 msec), the performance penalties almost doubled (the detailed results are not given here). Consequently, it is critical that the degradation in performance should be kept under control. In the next section, we present our preactivation strategy, which eliminates almost all performance penalty due to power management.

5.2 Processor Preactivation

Preactivation is a strategy that minimizes the performance impact of energy optimization. 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 mainly on preactivation of memory modules [30], [17] and I/O peripherals [4]. In this section, we demonstrate how preactivation of inactive processors can improve performance and energy behavior.

While the 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 Fig. 6a by t_s cycles (see Fig. 6b). In this section, we propose a preactivation 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 when it is required to participate in computation.

The ideal use of this approach is illustrated in Fig. 6c. 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 resynchronization. Consequently, when the processor is requested, it would have just finished the resynchronization period. An important issue here is to determine the exact point in the code to start resynchronization. This may not be trivial because resynchronization penalty is given in terms of cycles and it needs to be reexpressed in terms of loop iterations since this (i.e., loop iteration) is the only unit we can use (at source level) to insert activation/deactivation calls. Essentially, in order to preactivate a processor in the power-down state (for a specific loop nest), we need to determine an iteration (in the previous loop nest) before which the processor is activated. Let us assume that each iteration of this previous loop nest takes C cycles, and that the resynchronization 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 an execution profile such as the one shown in Fig. 6c.

The length of the execution profile in Fig. 6c is the same as that of Fig. 6a. Its energy consumption, on the other hand, is:

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

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: $\frac{t_s}{2} < t_i$. This condition is better (that is, easier to satisfy) than the one derived in the previous section from both the energy and performance viewpoints.

Figs. 6d and 6e illustrate, on the other hand, the scenarios where this ideal preactivation strategy does not happen. In Fig. 6d, 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 Fig. 6e, 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 resynchronization period has a fixed length, beyond which the processor is up and starts to consume energy. We see that processor preactivation can be an effective technique provided that it is done at an appropriate point during execution. There might be several reasons why it may not be possible to achieve the ideal preactivation. 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 in computation in the next loop) may have N iterations and our preactivation 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 [51]). This, in turn, can increase the code size, which may not be tolerated in some memory-constrained embedded environments. Second, the previous loop may not have a 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 a performance penalty. We implemented the preactivation strategy explained above with the SUIF compiler [1]. Its implementation is very similar to that of software-based data prefetching [36]. The main difference is that instead of identifying the next data to be accessed, we try to predict idleness.

To evaluate our processor preactivation strategy (that is, version 3), we performed another set of experiments. The results given in Fig. 9 (normalized with respect to version 1) indicate that the average 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 percent, 15.0 percent, 46.1 percent, 44.4 percent, 61.7 percent, and 59.8 percent, respectively. When we compare these results with the corresponding values given in the previous section, we see that preactivation is beneficial from an energy perspective too. We also observed that, except for one benchmark (`btrix`), the compiler was able to easily insert the preactivation calls. In `btrix`, in order to insert the preactivation calls, the compiler needed to split [51] two loop nests (using SUIF); this led to a 3 percent increase in the executable size and a 2 percent degradation in performance. To conclude, processor preactivation is beneficial from both energy and performance perspectives.

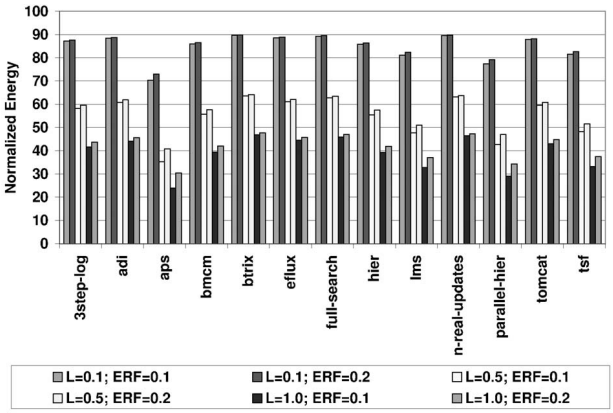

Fig. 9. Normalized energy consumption (version 3)). All the values are obtained assuming that a full leakage energy is consumed during the resynchronization periods.

In the next set of experiments, we modified two parameters: cache size and off-chip memory access energy. When we increase the cache size to 16KB (2-way, 32 bytes line size), we observed that there is not much change in energy gains. For example, the average energy savings in (L = 1.0; ERF = 0.1) and (L = 1.0; ERF = 0.2) are 62.4 percent and 59.3 percent, respectively. This is because of the fact that increasing cache size has two conflicting impacts. First, since a larger cache has also a larger per access dynamic energy and larger per cycle leakage energy, it tends to increase the contribution of cache to the overall energy. And, since we shut down cache (when it is not in use), a larger cache should increase energy savings. Second, a larger cache can capture more working set and reduce the number of misses. This, in turn, leads to a reduction in write-backs (to the cache), which is expected to reduce the energy savings due to a larger cache. These two impacts conflict with each other and, in case of the array-based codes in our experimental suite, they balance out each other. In order to approximate an embedded (on-chip) main memory, we conducted another group of experiments where the per access memory energy is reduced to one tenth of its original value. Since this reduces the contribution of the main memory energy to the overall energy budget (and we do not perform any energy optimization for the main memory), we observed an increase in energy savings due to processor shut down combined with preactivation. As a specific 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 percent and 63.6 percent, respectively.

5.3 Discussion

Our results presented so far demonstrate that employing adaptive parallelism and placing the unused processors (in a given loop nest) into a power-down state can bring large benefits over a strategy that keeps all processors in the active state during each nest execution. We also demonstrated that adopting an adaptive parallelization strategy can impact performance negatively due to frequent power-offs/ons of processors and their caches. One might argue that an alternative strategy, which determines the ideal number of processors for a given program and uses that processor size throughout the execution would also perform very well. An important advantage of such a strategy would be eliminating the extra latency/energy cost due to processor deactivations/reactivations across loop nest boundaries. Its main disadvantage is that the selected

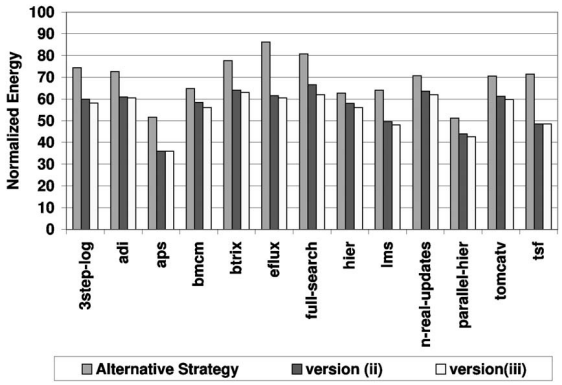

Fig. 10. Normalized energies for three different versions. The alternative strategy (the first bar for each benchmark code) determines the ideal number of processors considering the entire program and uses that processor size throughout the execution.

processor size may be suboptimal for some loop nests (since it is determined considering the entire application, i.e., not tuned based on individual loop nests).

To evaluate the performance of such an alternative strategy and to compare it with our strategies discussed above (i.e., versions 2 and 3), we performed another set of experiments. To find the ideal number of processors that will be used with this alternative strategy, we determined the energy consumption of each application under all possible processor sizes and, for each application, determined the ideal processor size from the energy perspective. Fig. 10 gives the normalized energy consumption for this alternative strategy (with L = 0.5 and ERF = 0.1). The results for versions 2 and 3 are reproduced here for ease of comparison. As before, all results are normalized with respect to version 1. We can observe from these results that versions 2 and 3 generate better energy results than the alternative strategy. This is because, although the alternative strategy does not incur any performance penalty, its energy behavior is not very good. This is a direct result of working with a single processor size throughout the execution. In addition, the energy-delay product of this alternate strategy is 6.5 percent worse than that achieved by our approach, when averaged over all benchmarks. That is, our approach outperforms the alternate strategy from both energy and energy-delay perspectives. It should also be noticed that applications are becoming increasingly large and complex. Consequently, one may expect even larger variances across the processor demands of different nests in the future (where we have hundreds of nests), which means that adaptively changing the number of processors will be even more important.

Another important issue that we want to discuss is the sensitivity of the results to the data set sizes. It is to be noted that, in array-based applications, data access pattern and communication pattern are dependent on the size of the data to be processed but not on contents of the data. In fact, in our simulations, when we make experiments with a certain input size, the data set (i.e., data contents) used in profiling and real execution were different and we observed that this did not make any difference, as compared to the case where the same data set is used for both profiling and actual execution. However, when we changed the size of the input, this made a difference in selection of the optimal number of processors used for loop nests. In particular, when we increase the number of processors, our approach tends to use larger processor sizes for the nests. Specifically, the average number of processors per nests increased by 16.8 percent when we doubled the input size. While this reduced our energy savings a bit, we were still able to achieve significant reductions in energy consumption.

6 Optimizing under Multiple Constraints

6.1 ILP Formulation

In the previous section, we presented an adaptive parallelization strategy that saves energy by exploiting the maximum loop-level parallelism available. In many cases, maximizing loop parallelism may not be the only possible objective function to target. In particular, in many energy-sensitive embedded environments, the objective of compilation can be different from just minimizing the energy consumption under the best parallelism. Consequently, in this section, we present a general strategy based on integer linear programming (ILP) for compiling a given application for an on-chip multiprocessor under multiple constraints.

ILP provides a set of techniques that solve those optimization problems in which both the objective function and constraints are linear functions and the solution variables are restricted to be integers. The zero-one ILP (ZILP) is an ILP problem in which each (solution) variable is restricted to be either zero or one [39]. Prior work [7] used an ILP-based approach to optimize code under performance and size (code length) constraints. Our focus here is on energy-performance trade offs.

Our approach has two phases: a profiling phase and an optimization phase. In the profiling phase, we run each loop nest of the application being optimized for each possible processor size and record the energy consumption and execution time. However, in profiling a particular loop nest, we also execute all the preceding loop nests so that we can capture the impact of internest data reuse. In other words, the collected profile data reflects the cache hit/miss pattern of an actual execution. These data are then fed to the second phase where we use integer linear programming (ILP) to determine the number of processors for each loop nest, taking into account the objective function and the compilation constraints. Therefore, the number of processors that will be used for executing each loop nest in the final optimized code is decided at compile time. However, the processor deactivations/reactivations (that is, placing a processor and its caches into a sleep state and later transitioning them to active state) take place at runtime.

In more detail, our optimization strategy is built upon the idea of precomputing the energy consumption and execution time of multiple versions of each loop nest with different number of processors and storing them in a table. To be specific, if we have M different loop nests in a given array-intensive application and N different processors in our on-chip multiprocessor, we prepare two tables (one for energy values and the other one for execution cycles) in which each loop nest has N entries (that is, a total of NM energy values and NM execution time values). An entry i, j in these tables gives the energy consumption (or execution cycles) when loop nest i is executed using j processors. While in this paper we experimented with an architecture that contains eight processors, it is not difficult to modify this strategy to work with a machine with a different number of processors. After building such a table, our approach takes into account the energy and performance constraints (given as input) and determines the ideal

number of processors for each loop nest. It should be noted that the number of processors selected for two consecutive loop nests might be different from each other and, in switching from one processor size to another, the associated time and energy overheads need to be accounted for. To select the processor sizes for loop nests, our approach formulates the problem as a ZILP and uses a publicly-available ILP solver [45] (where we explicitly encode the zero-one conditions).

Let us first make the following definitions, assuming that $1 \leq i \leq M$ and $1 \leq j \leq N$, where N is the total number of processors and M is the number of nests in the code being optimized:

- $E_{i,j}$: the estimated energy consumption for loop nest i when j processors are used to execute it.
- $X_{i,j}$: the estimated execution time for loop nest i when j processors are used to execute it.

We use zero-one integer variables $s_{i,j}$ to indicate whether j processors are selected for executing loop nest i or not. Specifically, if $s_{i,j}$ is 1, this means that j processors are selected to execute nest i in the final solution. Obviously,

$$\sum_{j=1}^{N} s_{i,j} = 1 \tag{1}$$

should be satisfied for each nest i. Using $E_{i,j}$, $X_{i,j}$, and $s_{i,j}$, the total energy consumption (E) and total execution time (X) for the application can be expressed as:

$$E = \sum_{i=1}^{M} \sum_{j=1}^{N} s_{i,j} E_{i,j}, \tag{2}$$

$$X = \sum_{i=1}^{M} \sum_{j=1}^{N} s_{i,j} X_{i,j}. \tag{3}$$

It should be noted, however, that these energy and execution time calculations do not include any overhead for changing the number of processors between nest boundaries.

As mentioned earlier, in our experiments, we assumed that a fixed (constant) amount time (denoted R) is required to make an inactive processor (i.e., a processor that has been placed into the sleep state along with its caches) active. We can compute the execution time overhead as follows:

$$n_{i,j} \geq s_{i,j} - \sum_{k=j}^{N} s_{i-1,k}, \tag{4}$$

$$y_i = \sum_{j=1}^{N} n_{i,j}, \tag{5}$$

$$Y = R \sum_{i=2}^{M} y_i, \tag{6}$$

where $2 \leq i \leq M$ and $1 \leq j \leq N$. In this formulation, $n_{i,j}$ and y_i are zero-one variables. Expression (4) restricts $n_{i,j}$ to be 1 if loop nest i has j active processors and loop nest $i-1$ has less than j active processors; otherwise, $n_{i,j}$ is restricted to be 0. Expression (5), on the other hand, sets y_i to 1 if loop nest i has more active processors than loop nest $i-1$. Note that only when loop nest i uses more processors than loop nest $i-1$ do we incur a resynchronization penalty in going from loop nest $i-1$ to loop nest i. Finally, (6) computes the total time overhead for activating processors between loop nest boundaries. Note that R is constant (20 msec in our experiments). It is also to be noted that, if loop nest i has less active processors than nest $i-1$, we have $n_{i,j} \geq 0$. This implies that $n_{i,j}$ can be either 0 or 1. However, since $n_{i,j}$ is a part of the cost function ($X+Y$) we want to minimize, the ILP solver always sets $n_{i,j}$ to 0 in this case.

Suppose that, in going from one loop nest to another, we need to activate new processors. These new processors plus the ones used to execute the former loop nest are considered to consume leakage energy for the time period of R. The energy consumption, including overheads, can be computed as follows:

$$t_{i,j} = \sum_{k=1}^{j} s_{i,k}, \tag{7}$$

$$z_i = \sum_{j=1}^{N} t_{i,j} - 1, \tag{8}$$

$$u_i \geq z_{i-1} - z_i, \tag{9}$$

$$v_i \geq 1 - u_i, \tag{10}$$

$$U = \sum_{i=2}^{M} u_i, \tag{11}$$

$$Z = r(U + \sum_{i=1}^{M-1}((1-v_i)\sum_{j=1}^{N} j s_{i,j})), \tag{12}$$

where $1 \leq i \leq M$ and $1 \leq j \leq N$, except for (9), where $2 \leq i \leq M$. Here, r is the leakage energy consumption of a single processor during the time period of R and is constant. In this formulation, $t_{i,j}$ is a zero-one variable. Expression (7) sets $t_{i,j}$ to 1 if loop nest i has j or less than j active processors. Expression (8) gives the number of inactive (powered-down) processors at loop nest i. Expression (9) forces u_i to take 0 or a positive value and determines the number of processors to be activated in going from loop nest $i-1$ to loop nest i. Expression (10) defines v_i, which is used to decide whether there is an activation period between two loop nests. If u_i is 0, then v_i will be 1, meaning that there is no activation period. Otherwise, if u_i is greater than 0, then v_i will be 0 (this is again due to the minimization problem), indicating an activation period. Expression (11) gives the total number of processors that need to be activated between loop nest boundaries. Finally, (12) gives the total leakage energy overhead for the processors activated between loop nest boundaries and the processors already active during activation periods.

Taking into account these energy and performance overheads, $X+Y$ gives the execution time, including overheads, and $E+Z$ gives the energy consumption, including overhead energy. It should also be noted that expressions for calculating time overhead and energy overhead are different in a sense that, while we overlap the time overhead for activated processors between loop nest boundaries, we consider each activated processor to consume leakage energy during an activation period.

In some cases, the optimization problem involves constraints or an objective function built from energy consumptions of different hardware components. One example would be minimizing main memory energy only (instead of the overall system energy) under a specific execution time and a specific overall energy consumption constraints. To capture

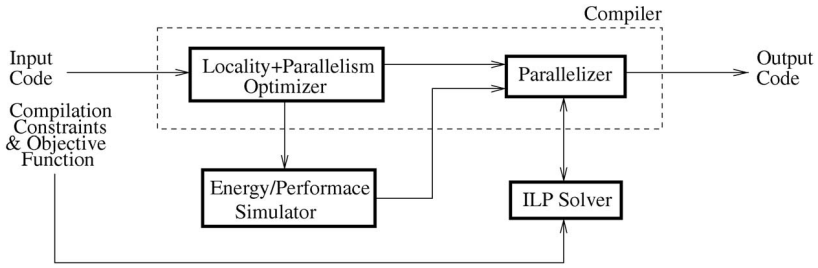

Fig. 11. Our optimization system. Note that the parallelization is performed based on the input provided by the ILP solver.

the energy breakdown across different components, we use $E^d_{i,j}$, $E^{dc}_{i,j}$, $E^{ic}_{i,j}$, and $E^m_{i,j}$ to denote (for loop nest i executing using j processors) the estimated energy consumptions in datapath, data cache, instruction cache, and main memory, respectively. If these are the only hardware components of interest, then we have:

$$E_{i,j} = E^d_{i,j} + E^{dc}_{i,j} + E^{ic}_{i,j} + E^m_{i,j}.$$

We can find the overall main memory (E^m), instruction cache (E^{ic}), data cache (E^{dc}), and datapath energies (E^d) as: $E^m = \sum_i \sum_j E^m_{i,j} s_{i,j}$, $E^{ic} = \sum_i \sum_j E^{ic}_{i,j} s_{i,j}$, $E^{dc} = \sum_i \sum_j E^{dc}_{i,j} s_{i,j}$, and $E^d = \sum_i \sum_j E^d_{i,j} s_{i,j}$.

6.2 Experiments and Results

To evaluate our ILP-based optimization strategy, we performed experiments with our array-intensive benchmarks. Fig. 11 shows the structure of our optimization framework. The input code is first optimized using classical data locality optimizations as well as loop transformations that enhance loop-level parallelism (e.g., loop skewing) [51]. Then, this optimized code is fed to our simulator. The simulator simulates each loop nest using all possible processor sizes and records energy consumption (both total and component-wise) and execution time for each loop nest. After that, this information, along with the code itself, is fed to the parallelization module (shown as "parallelizer" in the figure). This module constructs an ILP formulation (in a form suitable for the solver) and passes it to the ILP solver. The solver also takes the compilation constraints and the objective function as input. The parallelization module then obtains the solution from the solver and parallelizes the application (i.e., each loop nest) accordingly. If it is not possible to parallelize the application under given constraints and objective function, an error message is signaled. If desired, the user can relax some constraints and retry compilation.[2]

We first focus on `btrix` and present results when this application is compiled under different energy and performance constraints. Let us assume for now that the overhead of dynamically changing the number of processors is zero. We consider different compilation strategies, whose objective functions and performance/energy constraints (if any) are given in the second and third columns, respectively, of Fig. 13. First, let us focus on two simple cases: Case-I and Case-II. In Case-I, we try to minimize execution time; note that this is the classical objective of a performance-oriented compilation strategy. We see from columns four through ten of Fig. 13 that, in this case, the average number of (selected) processors per loop nest is 4.00. Case-II represents the other end of the spectrum where we strive for minimizing the energy consumption (without any performance concerns). The average number of processors per loop nest is 2.00. The last two columns in Fig. 13 give the energy consumption (in microjoules) and execution time (in msecs) for each scenario. We observe that optimizing only for performance results in an energy consumption which is 27.1 percent higher than the best energy-optimized version. Similarly, optimizing only for energy consumption leads to an execution time which is 20.0 percent higher than the best performance-optimized version. These results demonstrate that parallelization for energy and performance can generate very different results.

Let us now focus on more interesting compilation strategies. Case-III and Case-IV correspond to minimizing energy consumption under performance constraints. We note that the processor sizes selected in these cases are different from those selected for Case-II. The average number of processors per loop nest are 2.57 and 3.28 for Case-III and Case-IV; that is, including a performance constraint in the compilation increases the average number of processors used. In addition, we see that a more stringent performance constraint ($X \leq 31,000$) results in a higher number of processors being selected. Case-V corresponds to optimizing execution time under energy constraint. One can observe that, in this case, the number of processors selected for some loop nests is different from the corresponding numbers in Case-I. To sum up, when we consider Cases III, IV, and V, we see that including constraints during compilation can change the number of processors selected for executing loop nests. Note also that, without an ILP-based framework such as the one presented in this paper, it would be extremely difficult to decide on the number of processors for each loop nest when compilation scenarios such as Case-V, Case-II, or Case-III are encountered.

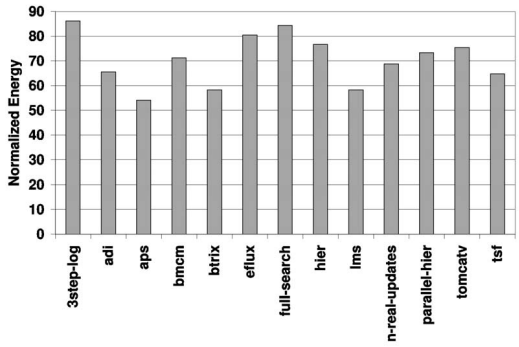

Fig. 12. Normalized energy consumptions under performance bounds.

2. As a feedback to the user, we return the constraint that led to the failure.

Case	Objective Function	Compilation Constraints	Selected Number of Processors							Energy Consumption	Execution Time
			N1	N2	N3	N4	N5	N6	N7		
Case-I	minimize X	none	2	1	7	6	1	3	8	58629	29219
Case-II	minimize E	none	2	1	2	2	1	2	4	46133	38183
Case-III	minimize E	$X \leq 33000$	2	1	4	2	1	3	5	51973	32986
Case-IV	minimize E	$X \leq 31000$	2	1	6	4	1	3	6	54018	30526
Case-V	minimize X	$E \leq 55000$	2	1	6	5	1	3	6	54211	30120
Case-VI	minimize $E + Z$	$X + Y \leq 33000$	2	1	2	2	1	3	3	52875	33205
Case-VII	minimize $E + Z$	$X + Y \leq 31000$	2	1	4	4	1	3	3	54644	30986
Case-VIII	minimize $X + Y$	$E + Z \leq 56000$	2	1	5	4	1	3	3	55216	30210

Fig. 13. Different compilation strategies, the selected number of processors, energy consumptions, and execution times (`btrix`). Note that different compilation objectives/constraints generate entirely different execution times and energy consumptions.

Up to this point, we have assumed that the overhead of dynamically changing the number of processors between loop nests is zero. The remaining cases in Fig. 13 correspond to scenarios with nonzero energy and performance overhead for reactivating an inactive processor. Specifically, we assumed a resynchronization penalty of 20 msec during which all processors (except the ones in low power) are assumed to consume leakage energy. We see from Cases VI, VII, and VIII that when the overhead is taken into account, the ILP solver tends to keep the number of processors the same between the neighboring loop nests as much as possible. In fact, in Cases VI and VIII, our approach employs only three different processor sizes as compared to the six processor sizes used in Case-I. However, we also see that even in these cases, optimizing for performance and optimizing for energy generate different number of processors for most of the loop nests.

In our final set of experiments, we evaluate how our approach optimizes all the codes in our experimental suite. Fig. 12 presents the energy consumption obtained using our approach under performance constraints (similar to Case-VI). Each bar here represents an energy consumption value "normalized" with respect to an "alternative strategy" that uses a fixed number of processors (i.e., the best processor size for the application in question) throughout the execution (under the "same maximum execution time constraint"). Note that to achieve the same execution time as our ILP-based approach, the mentioned alternative strategy employs a larger number of processors, leading to a much higher energy consumption. Specifically, the results given in this graph indicate that our approach improves energy behavior by 29.4 percent, compared to the strategy that fixes the number of processors throughout the execution. Therefore, we can conclude that tuning the number of processors according to the requirements of each loop nest is important to achieve minimum energy values under performance constraints.

7 Concluding Remarks

Based on the observation that in a given array-intensive code, not all the loop nests require the maximum number of processors in the on-chip multiprocessor, in this paper, we first 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 preactivation strategy. Our experiments with an eight-processor on-chip multiprocessor and different parameters (e.g., cache size, resynchronization overhead, and per access off-chip memory energy) indicated that our approach is successful in reducing energy consumption. We then presented an integer linear programming (ILP) based strategy for compiling a given array-intensive application in an on-chip multiprocessor under energy/performance constraints. Our strategy has two parts: profiling and ILP formulation. At the end of the optimization process, the compiler determines the most suitable number of processors that will be used in optimizing each nest in the code. Our preliminary results are very encouraging, indicating up to 54 percent savings in energy compared to a static scheme that uses the maximum number of available processors for each nest. We also demonstrated that our approach can optimize a given application targeting energy consumption of individual hardware components.

Acknowledgments

This work is supported in part by the US National Science Foundation Career Award #0093082. Parts of this material were presented at DAC '02 (Design Automation Conference) [23]. This paper augments the material in the DAC paper by providing an ILP formulation of the constrained optimization problem and an experimental evaluation.

References

[1] S.P. Amarasinghe, J.M. Anderson, M.S. Lam, and C.W. Tseng, "The SUIF Compiler for Scalable Parallel Machines," *Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computing,* Feb. 1995.

[2] R.I. Bahar and S. Manne, "Power and Energy Reduction via Pipeline Balancing," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 218-229, 2001.

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

[4] L. Benini, G. De Micheli, "System-Level Power Optimization: Techniques and Tools," *ACM Trans. Design Automation of Electronic Systems,* vol. 5, no. 2, pp. 115-192, 2000.

[5] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Extending Lifetime of Portable Systems by Battery Scheduling," *Proc. Design, Automation, and Test in Europe Conf.,* pp. 197-201, Mar. 2001.

[6] M. Berry et al., "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers," *The Int'l J. Supercomputer Applications,* 1988.

[7] F. Bodin, Z. Chamski, C. Eisenbeis, E. Rohou, A. Seznec, "GCDS: A Compiler Strategy for Trading Code Size against Performance in Embedded Applications," Technical Report RR-3346, INRIA, Rocquencourt, France, Jan. 1998.

[8] J.A. Butts and G. Sohi, "A Static Power Model for Architects," *Proc. Int'l Symp. Microarchitecture,* Dec. 2000.

[9] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook, "Adaptive Parallelism with Piranha," Technical Report 954, Yale Univ., Feb. 1993.

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

[11] Y. Chen, "Architectural Level Power Estimation for Systems-on-a-Chip," PhD thesis, Pennsylvania State Univ., May 1999.

[12] Chip Multiprocessing, http://industry.java.sun.com/javanews/stories/print/0,1797,32080,00.html, 2004.

[13] Chip Multiprocessing, ITWorld.Com, http://www.itworld.com/Comp/1092/CWSTO54343/, 2004.

[14] T.M. Conte, K.N. Menezes, S.W. Sathaye, and M.C. Toburen, "System-Level Power Consumption Modeling and Tradeoff Analysis Techniques for Superscalar Processor Design," *IEEE Trans. VLSI Systems,* vol. 8, no. 2, Apr. 2000.

[15] D.E. Culler and J.P. Singh, *Parallel Computer Architecture: A Hardware-Software Approach.* Morgan Kaufmann, 1999.

[16] "Design Methodologies Meet Network Applications" and "System on Chip Design," *Proc. Design Automation Conf. '02 Sessions,* June 2002.

[17] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M.J. Irwin, "DRAM Energy Management Using Software and Hardware Directed Power Mode Control," *Proc. Seventh Int'l Conf. High Performance Computer Architecture,* Jan. 2001.

[18] D. Duarte, N. Vijaykrishnan, and M.J. Irwin, "A Clock Power Model to Evaluate Impact of Architectural and Technology Optimizations," *IEEE Trans. VLSI,* to appear.

[19] K. Flautner, N. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," *Proc. Int'l Symp. Computer Architecture,* June 2002.

[20] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *J. Parallel & Distributed Computing,* vol. 5, no. 5, pp. 587-616, Oct. 1988.

[21] J.P. Halter and F. Najm, "A Gate-Level Leakage Power Reduction Method for Ultra-Low-Power CMOS Circuits," *Proc. IEEE Custom Integrated Circuits Conf.,* pp. 475-478, 1997.

[22] I. Kadayif, M. Kandemir, and U. Sezer, "An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors," *Proc. Design Automation Conf.,* June 2002.

[23] I. Kadayif, M. Kandemir, and M. Karakoy, "An Energy Saving Strategy Based on Adaptive Loop Parallelization," *Proc. Design Automation Conf.,* June 2002.

[24] I. Kadayif, I. Kolcu, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin, "Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption in Chip Multiprocessor," *Proc. Seventh Design Automation and Test in Europe Conf.,* Feb. 2004.

[25] M. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *Proc. Int'l Symp. Low Power Electronics and Design,* Aug. 1997.

[26] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *Proc. 28th Int'l Symp. Computer Architecture,* 2001.

[27] V. Krishnan and J. Torrellas, "A Chip Multiprocessor Architecture with Speculative Multi-Threading," *IEEE Trans. Computers,* special issue on multithreaded architecture, vol. 48, no. 9, Sept. 1999.

[28] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* p. 81, 2003.

[29] T. Kuroda and T. Sakurai, "Threshold-Voltage Control Schemes through Substrate-Bias for Low-Power High-Speed CMOS LSI Design," *J. VLSI Signal Processing Systems,* vol. 13, nos. 2/3, pp. 191-201, Aug. 1996.

[30] A.R. Lebeck, X. Fan, H. Zeng, and C.S. Ellis, "Power-Aware Page Allocation," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Nov. 2000.

[31] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, and A. Sivasubramaniam, "Leakage Energy Management in Cache Hierarchies," *Proc. 11th Int'l Conf. Parallel Architectures and Compilation Techniques,* 2002.

[32] J. Li, J. Martinez, and M. Huang, "The Thrifty Barrier: Energy-Efficient Synchronization in Shared-Memory Multiprocessors," *Proc. High Performance Computer Architecture,* pp. 14-23, 2004.

[33] L. Macchiarulo, E. Macii, and M. Poncino, "Low-Energy Encoding for Deep-Submicron Address Buses," *Proc. Int'l Symp. Low Power Electronics and Design,* Aug. 2001.

[34] MAJC-5200, http://www.sun.com/microelectronics/MAJC/5200wp.html, 2004.

[35] D. Marculescu, "Profile-Driven Code Execution for Low Power Dissipation," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 253-255, 2000.

[36] T. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," PhD thesis, Stanford Univ., Computer Systems Laboratory, Mar. 1994.

[37] MP98: A Mobile Processor, http://www.labs.nec.co.jp/MP98/top-e.htm, 2004.

[38] B.A. Nayfeh, L. Hammond, and K. Olukotun, "Evaluating Alternatives for a Multiprocessor Microprocessor," *Proc. 23rd Int'l Symp. Computer Architecture,* pp. 66-77, 1996.

[39] G. Nemhauser and L. Wolsey, *Integer and Combinatorial Optimization.* New York: Wiley-Interscience Publications, John Wiley & Sons, 1988.

[40] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single Chip Multiprocessor," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 2-11, 1996.

[41] C. Polychronopoulos and D. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers,* vol. 36, pp. 1425-1439, 1987.

[42] D. Rypl and Z. Bittnar, "Mesh Generation Techniques for Sequential and Parallel Processing," *Aspects in Modern Computational Structural Analysis,* pp. 257-276, 1997.

[43] R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes, "The Energy Efficiency of CMP vs. SMT for Multimedia Workloads," *Proc. 18th Ann. Int'l Conf. Supercomputing,* pp. 196-206, 2004.

[44] R.K. Scannell, "A 480-MFLOP MCM Based on the SHARC DSP Chip Breaks the MCM Cost Barrier," *Proc. Int'l Conf. Multichip Modules,* 1996.

[45] H. Schwab, lp_solve Mixed Integer Linear Program Solver, ftp://ftp.es.ele.tue.nl/pub/lp_solve/, 2004.

[46] W.-T. Shiue and C. Chakrabarti, "Memory Exploration for Low-Power Embedded Systems," *Proc. Design Automation Conf.,* 1999.

[47] T. Simunic, L. Benini, P.W. Glynn, and G. De Micheli, "Dynamic Power Management for Portable Systems," *Proc. MOBICOM,* pp. 11-19, 2000.

[48] TI Military Multimedia Video Processor (MVP) 320C8X, http://www.ti.com/sc/docs/products/military/processr/320c8x.htm, 2004.

[49] I. Verbauwhede and C. Nicol, "Low Power DSPS for Wireless Communications," *Proc. Int'l Symp. Low Power Electronics and Design,* 2000.

[50] N. Vijaykrishnan, M. Kandemir, M.J. Irwin, H.Y. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using Simplepower," *Proc. Int'l Symp. Computer Architecture,* June 2000.

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

[52] S. Wilton and N. Jouppi, "Cacti: An Enhanced Cache Access and Cycle Time Model," *IEEE J. Solid-State Circuits,* May 1996.

[53] W. Zhang, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, D. Duarte, and Y. Tsai, "Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction," *Proc. 34th Ann. Int'l Symp. Microarchitecture,* Dec. 2001.

[54] V. Zivojnovic, J. Velarde, and C. Schlager, "DSPstone: A DSP-Oriented Benchmarking Methodology," *Proc. Fifth Int'l Conf. Signal Processing Applications and Technology,* Oct. 1994.

Ismail Kadayif received the BSc degree in 1991 from the Department of Control and Computer Engineering, Istanbul Technical University, Istanbul, Turkey. He received the MSc degree from the Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois in 1997. He received the PhD degree from the Department of Computer Science and Engineering at the Pennsylvania State University in 2003. He has been an assistant professor at the Canakkale Onsekiz Mart University, Turkey, since August 2003. He is interested in high-level compiler optimizations, power-aware compilation techniques, and low-power computer architectures. He is a student member of the IEEE and the IEEE Computer Society.

Mahmut Kandemir received the BSc and MSc degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the PhD degree from Syracuse University, Syracuse, New York, in electrical engineering and computer science in 1999. He has been an associate professor in the Computer Science and Engineering Department at the Pennsylvania State University since July 2004. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a recipient of a US National Science Foundation Career Award. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Guilin Chen is a PhD student in the Department of Computer Science and Engineering at the Pennsylvania State University. His current research interests include compilers, low-power architectures, Java virtual machines, and reliable computing. Before coming to Penn State, he studied in the Computer Science Department at Fudan University, China, where he received the BS degree in 1998 and the MS degree in 2001. From 1998 to 2001, he worked as a research assistant at the Institute of Parallel Processing at Fudan University. He is a student member of the IEEE and the IEEE Computer Society.

Ozcan Ozturk received the BSc degree in computer engineering from Bogazici University, Istanbul, Turkey, in 2000 and the MS degree in computer engineering from University Of Florida, Gainesville, in 2002. He is currently pursuing the PhD degree in computer science and engineering at the Pennsylvania State University. His research interests are in the areas of on-chip multiprocessing, power-aware architectures, and compiler optimizations. He is a student member of the IEEE and the IEEE Computer Society.

Mustafa Karakoy received the BSc degree in control and computer engineering from Istanbul Technical University in 1988, and the MPhil degree in computer science from Cranfield University in 1996. Currently, he is a PhD student in the Department of Computing at Imperial College, London. He also works for Medicsight plc., a medical software company based in London. His research interests include neural networks, pattern recognition, component-based programming, and parallel computing.

Ugur Sezer received the BS degree in electrical and electronics engineering from the Dokuz Eylul University, Izmir, Turkey, in 1990, the MS degree in computer engineering from Northeastern University, Boston, in 1994, and is currently pursuing the PhD degree in computer engineering at the University of Wisconsin-Madison. His main interests are in real-time multimedia communication, image and video processing, media and file servers, and power-aware software optimizations. He is a member of the IEEE, ACM, SNIA, and SMPTE.