Exploiting Processor Workload Heterogeneity for Reducing Energy Consumption in Chip Multiprocessors

I. Kadayif
Dept. of Computer Engr.
Canakkale Onsekiz Mart Univ.
Canakkale, Turkey
kadayif@comu.edu.tr

M. Kandemir
CSE Department
Pennsylvania State University
Univesity Park, PA 16802, USA
kandemir@cse.psu.edu

I. Kolcu
Computation Department
UMIST
Manchester M60 1QD, UK
ikolcu@umist.ac.uk

Abstract

Advances in semiconductor technology are enabling designs with several hundred million transistors. Since building sophisticated single processor based systems is a complex process from design, verification, and software development perspectives, the use of chip multiprocessing is inevitable in future microprocessors. In fact, the abundance of explicit loop-level parallelism in many embedded applications helps us identify chip multiprocessing as one of the most promising directions in designing systems for embedded applications. Another architectural trend that we observe in embedded systems, namely, multi-voltage processors, is driven by the need of reducing energy consumption during program execution. Practical implementations such as Transmeta's Crusoe and Intel's XScale tune processor voltage/frequency depending on current execution load. Considering these two trends, chip multiprocessing and voltage/frequency scaling, this paper presents an optimization strategy for an architecture that makes use of both chip parallelism and voltage scaling. In our proposal, the compiler takes advantage of heterogeneity in parallel execution between the loads of different processors and assigns different voltages/frequencies to different processors if doing so reduces energy consumption without increasing overall execution cycles significantly. Our experiments with a set of applications show that this optimization can bring large energy benefits without much performance loss.

1. Introduction

Rising development costs motivate computer architecture companies to design fewer systems-on-chip, but to make each one they do design more flexible and programmable. Doing so makes it possible to reuse designs to take advantage of economies of scale and shorten time-to-market. Moreover, programmability allows companies to keep products in the market longer, boosting integrated profits.

High-performance embedded processors have traditionally relied mainly on clock frequency and superscalar instruction issue to boost performance. While frequency and superscalarity have served the industry well and will continue to be used, we believe that they have limitations that will diminish the gains they will deliver in the future. The gains in operating frequencies, which have historically come at a rate of about 35 percent per year, are attributable to two major factors: semiconductor feature scaling and deeper pipelining. But each of these factors is approaching the point of diminishing returns. Similarly, superscalar processing is nearing its limits, mainly due to the exponential increase in complexity in dispatch logic with increasing issue width. In addition, superscalar processing is limited by the inherent instruction-level parallelism in the code. Although VLIW implementations are less complex than their superscalar counterparts (since most of execution decisions are made by the compiler), they still employ power-hungry components and are limited by the available instruction-level parallelism. It should also be noted that both superscalar and VLIW architectures are not efficient from an energy consumption viewpoint. Therefore, it is not clear whether current architectures will be sufficient for meeting continuously increasing power and performance demands of applications.

These observations motive system designers to investigate different architectures. When one looks at computer architecture industry today, two different trends in system design can easily be observed: *on-chip multi-processing* and *multi-voltage processors*. On-chip multi-processors take advantage of high-level, coarse-grain parallelism that exists due to the natural independence of separate program fragments (e.g., functions and loops). As compared to superscalar and VLIW architectures, they are much more suitable for array-intensive embedded applications. Another advantage of using an on-chip multiprocessor, instead of a more powerful and sophisticated uniprocessor, is that there is less difficulty in designing a smaller, less complex chip. This also speedups chip verification and validation. Thus, time required to put the chip in the market becomes shorter. One can see several examples of on-chip multi-processing today in both academia and industry. For example, the four-core Hydra from Stanford University [14] is built around Integrated Device Technology Inc.'s RC32364 processor, which uses a 0.25-micron process, and runs at 250 MHz. As manufacturing processes keep getting refined, it becomes even easier to replicate the core several times on a single die. The MAJC architecture from Sun Microsystems [11] allows one to four processors to share the same die, and for each to run separate threads. Each processor is limited to four functional units (each of which are able execute both integer and floating point operations, making the MAJC architecture more flexible). Another example of an on-chip multiprocessor from industry is the Power4 processor from IBM [15], where two processors are placed into the same die.

The second trend, multi-voltage processors, is mainly driven by the need to reduce energy consumption dur-

1530-1591/04 $20.00 (c) 2004 IEEE

ing program execution. Practical implementations such as Transmeta's Crusoe [10] and Intel's XScale [8] scale processor voltage/frequency depending on execution load. Observing that one rarely needs an application to exercise a processor's maximum performance and the unused extra performance usually represents wasted energy, Crusoe designers try to match the operating level of the processor (in terms of voltage and frequency) to the performance requirements of the application being executed. Depending on the voltage regulator, a Crusoe processor can change its voltage in steps of 25mV and its frequency in steps of 33MHz.

Considering the continuously pressing power and performance demands, we can expect these two techniques to be co-exist in the future embedded architectures. Specifically, we believe that future architectures will be based on on-chip multi-processors, where each on-chip processor can be individually voltage/frequency scaled. Considering such an architecture, this paper investigates the energy/performance tradeoffs in parallelizing array-intensive applications taking into account the possibility that individual processors can operate in different voltage/frequency levels. In assigning voltage levels to processors, we make use of compiler analysis that reveals heterogeneity between the loads of different processors in parallel execution. Our experiments with a set of applications show that the proposed optimization can bring large energy benefits without much performance penalty.

The rest of this paper is organized as follows. The next sections describes our chip multiprocessor. Section 3 discusses why we may be experiencing load imbalance across on-chip processors at runtime. Section 4 discusses the necessary compiler analysis for determining workloads (on a loop nest basis) of individual processors participating in parallel computation. Section 5 discusses additional optimizations to further enhance our power savings. Section 6 describes our implementation, experimental platforms, and presents performance and energy numbers. Section 7 presents our concluding remarks.

2. Chip multiprocessor architecture and execution model

The chip multiprocessor we consider here is a shared-memory architecture; that is, the entire address space is accessible by all processors. Each processor has a private L1 cache, and shared memory is assumed to be off-chip. Optionally, we may include a (shared) L2 cache as well. Note that several architectures from academia and industry fit in this description [1, 14, 11, 12]. We keep the subsequent discussion simple by using a shared bus as the interconnect (though one could use fancier/higher bandwidth interconnects as well). We also use the MESI [19] protocol (the choice is orthogonal to the focus of this paper) to keep the caches coherent across the CPUs. We assume that voltage level and frequency of each processor in this architecture can be set independently of the others, and this is the main mechanism through which we save power. This paper focuses on a single-issue, five-stage (instruction fetch (IF), instruction decode/operand fetch (ID), execution (EXE), memory access (MEM), and write-back (WB) stages) pipelined datapath for each on-chip processor. Currently, this is the only architectural model for which our compiler estimates processor workload.

Note that progress in VLSI technology has allowed chipmakers to pack millions of transistors in a single die. Rather than throwing all these resources into a single, powerful processing core and making this core very complex to design and verify, chip-multiprocessors consisting of several simpler processor cores can offer a more cost-effective and simpler way of exploiting these higher levels of integration. Chip multiprocessors also offer a higher granularity (thread/process level) at which parallelism in programs can be exploited by compiler/runtime support, rather than leaving it to the hardware to extract the parallelism at the instruction level on a single (larger) multiple-issue core. All these compelling reasons motivate the trends toward chip multiprocessor architectures, and there is clear evidence of this trend in the several commercial offerings and research projects [1, 14, 11, 12].

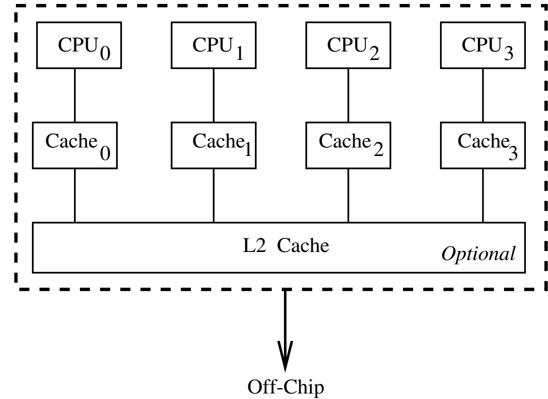

Figure 1. Chip multiprocessor under consideration.

Our application execution strategy can be summarized as follows. We focus on array-based applications that are constructed from loop nests. Typically, each loop nest in such an application is small but executes a large number of iterations and accesses/manipulates large datasets (typically multidimensional arrays). We employ a loop nest based application parallelization strategy. More specifically, each loop nest is parallelized independently of the others. In this context, parallelizing a loop nest means distributing its iterations across processors and allowing processors to execute their portions in parallel. For example, a loop with 1000 iterations can be parallelized across 10 processors by allocating 100 iterations to each processor. We also assume that after each loop nest execution, all processors get *synchronized* before they start executing the next loop nest. Note that dropping this requirement would necessitate a sophisticated compiler analysis to identify the cases under which a processor that finishes its portion of iterations from the previous loop nest can go ahead and start executing its portion from the next loop nest without waiting for the others. Nevertheless, in our experiments to be presented later, we also evaluate such an alternative strategy.

There are many proposals for power management of a dynamic voltage scaling-capable processor. Most of them are at operating system level and are either task-based [13, 17] or interval-based [21, 5]. While some proposals aim at reducing energy without compromising performance, a recent study by Grunwald et al [6] observed noticeable performance loss for some interval-based algorithms using actual measurements. The existing compiler based studies such as [7, 16] target single processor architectures. In comparison, our work targets at a chip multiprocessor based environment.

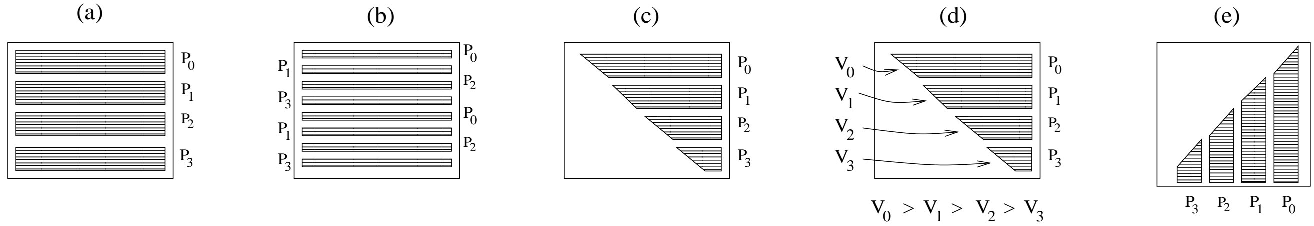

Figure 2. Different array accesses imposed by different iteration assignments (the array is assumed to be row-major).

3. Load imbalance in parallel execution

We can broadly divide loop nest parallelization techniques into two categories: *static* and *dynamic*. In the static case, the compiler (or the user) decides a suitable parallelization strategy for each loop nest at compile time. The idea is to assign each loop iteration to a processor. There are at least two ways of doing this. In block assignment, a group of consecutive loop iterations are assigned to the same processor. Since such iterations typically access data stored in consecutive memory locations, this type of assignment can also be expected to be data locality friendly. In cyclic assignment, the iterations assigned to processors are interleaved using some stride. While this type of assignment is known to be good from a load balance viewpoint, it generally exhibits poor data locality. Consider, as an example, the loop nest shown below and the array reference in it:

```
for i: 1..1024
  for j: 1..1024
      ..X[i,j]..
```

Assuming that only the `i`-loop is parallelized across four processors (P_0 through P_3), Figure 2(a) illustrates how array `X` is accessed by the processors when block iteration assignment is used. In this assignment, each processor executes 256×1024 iterations, and accesses a group of consecutive rows of the array as depicted in Figure 2(a). However, it is also possible to parallelize this loop (`i`) by distributing its iterations cyclicly across processors using some regular stride. For example, we can give the first 128×1024 iterations to the first processor, the next 128×1024 to the second one and so on, and when we give its quota to the last processor, we can repeat the whole process (until all loop iterations have been assigned) starting over with the first processor. Figure 2(b) shows how array `X` is accessed by the processors under this cyclic iteration assignment scheme. Note that the cyclic iteration distribution is flexible in the sense that it can work with any stride. For example, instead of using 128×1024 iteration chunks, we could have easily used 16×1024 or even 1×1024 iteration chunks.

In comparison, in a dynamic parallelization strategy, the assignment of iterations to processors is performed dynamically during the course of execution by a central controller. Typically, this controller gives a new set of loop iterations to a processor when that processor is done with executing its current set of assigned iterations. While the dynamic strategy is expected to balance the workloads of processors better than static strategies (as it can take runtime constraints into account), it also incurs a much higher runtime cost — in terms of both execution cycles and power consumption — (as compared to the static parallelization schemes) since decisions regarding iteration assignments are made at runtime. Therefore, our focus in this study is on static loop nest parallelization.

Consider now the following loop nest:

```
for i: 1..1024
  for j: i..1024
      ..X[i,j]..
```

While this loop nest is similar to the previous one considered above, there is one significant difference: the lower bound of the inner loop (`j`) is `i` (instead of 1). Figure 2(c) shows how the four processors access the array in question when block iteration assignment is employed. Clearly, there is a significant *load imbalance* across the processors. Assuming that each iteration of this loop nest has the same cost (in terms of execution cycles) and all processors should synchronize following the execution of the nest, there is not any advantage for the processors with the light load to finish their set of iterations as soon as possible. Instead, they can delay their executions (by reducing their frequencies) and lower their voltages to save energy while making sure that their execution does not take more time than that of the processor with the largest load (operating with the highest voltage level). Figure 2(d) illustrates such a voltage assignment, assuming that V_0 is the highest voltage level available. The work presented in this paper performs such a voltage-to-processor assignment for each loop nest of a given array-based application. In a sense, *in our framework the job of the compiler is not just to decide which loop iterations should be assigned to which processors but also which supply voltage/frequency each processor needs to use. Our objective is to save as much power as possible without incurring much performance penalty.*

At this point, someone might claim that it would be better in this case (Figure 2(c)) to use cyclic assignment instead of block assignment as this would eliminate the load imbalance problem introduced by the latter to a large extent. However, this may not be a viable option in general. Consider, for example, the scenario depicted in Figure 2(e), where the direction of parallelization is reversed (due to data dependences for example). In this case, cyclic assignment would be very costly in terms of data locality (cache behavior), assuming that the array in question is stored as row-major. Considering the fact that off-chip memory accesses are getting more and more expensive in terms of processor cycle times, one may not want to degrade data locality.

4. Compiler support

As mentioned earlier, the compiler's job in our setting is to assign not only iterations to processors but also come up with a suitable voltage level for each processor. To do this, the compiler needs to estimate the workload of each processor and match it with an appropriate voltage/frequency level. Without loss of generality, we assume that there are

s voltage/frequency levels available to the compiler. Our compiler-based approach proceeds as follows:

- **Parallelization Step.** In this step, the compiler parallelizes an application in a loop nest basis. That is, each loop nest is parallelized independently considering the intrinsic data dependences it has. Since we are targeting a chip multiprocessor, our parallelization strategy tries to achieve (for each nest) outer-loop parallelism to the best extent possible. In other words, we parallelize the outermost loop (in the nest) that carries no data dependence. Our baseline results are obtained using this parallelization strategy. Later in our experiments, we change our parallelization strategy to conduct a sensitivity analysis.

- **Processor Load Estimation.** In this step, the compiler estimates the load of each processor in each nest. To do this, it performs two calculations: (a) iteration count estimation and (b) per-iteration cost estimation. Since in most array-based embedded applications bounds of loops are known before execution starts, estimating the iteration count for each loop nest is not very difficult. The challenge is in determining the cost (in terms of execution cycles) of a single iteration (for a given loop nest). Since the processors employed in our chip multiprocessor are simple single-issue cores, our cost computation is closely dependent on the number and types of the assembly instructions that will be generated for the loop body. Specifically, we associate a base execution cost with each type of assembly instruction. In addition, we also estimate the number of cache misses. Since loop-based embedded applications exhibit very good instruction locality (as they spend most of their execution cycles within loop nests and there are not too many conditional-if executions), we focus on data cache and estimate data cache misses using the method proposed by Carr et al [2]. An important issue is to estimate (at the source level) what assembly instructions will be generated for the loop body in question. We attack this problem as follows. The constructs that are vital to the studied codes include a typical loop, a nested loop, assignment statements, array references, and scalar variable references within and outside loops. Our objective is to estimate the number of assembly instructions of each type associated with the actual execution of these constructs. To achieve this, the assembly equivalents of several codes were obtained using our back-end compiler (a variant of gcc) with the O2-level optimization. Next, the portions of the assembly code were correlated with corresponding high-level constructs to extract the number and type of each instruction associated with the construct. In order to simplify the correlation process and to partially isolate the impact of instruction choice due to low-level optimizations, the assembly instructions with similar functionality and energy consumption are grouped together. For example, both branch-if-not-equal (bne) and branch-if-equal (beq) are grouped as a generic branch instruction (denoted bra).

To illustrate our parameter extraction process in more detail, we focus on some specifics of the following example constructs. First, let us focus on a loop construct. Each loop construct is modeled to have a one-time overhead to load the loop index variable into a register and initialize it. Each loop also has an index comparison and an index increment (or decrement) overhead whose costs are proportional to the number of loop iterations (called trip count or trip). From correlating the high-level loop construct to the corresponding assembly code, each loop initialization code is estimated to execute one load (lw) and one add (add) instruction (in general). Similarly, an estimate of trip+1 load (lw), store-if-less-than (stl), and branch (bra) instructions is associated with the index variable comparison. For index variable increment (resp. decrement), $2\times$trip addition (resp. subtraction) and trip load, store, and jump instructions are estimated to be performed.

Next, we consider extracting the number of instructions associated with array accesses. First, the number and types of instructions required to compute the address of the element are identified. This requires the evaluation of the base address of the array and the offset provided by the subscript(s). Our current implementation considers the dimensionality of the array in question, and computes the necessary instructions for obtaining each subscript value. Computation of the subscript operations is modeled using multiple shift and addition/subtraction instructions (instead of multiplications) as this is the way our back-end compiler generates code when invoked with the O2 optimization flag. Finally, an additional load/store instruction was associated to read/write the corresponding array element. Note that these correlations between high-level constructs and low-level assembly instructions are a first-level approximation for our simple architecture and array-dominated codes with the O2-level optimization and obtained through extensive analysis of a large number of code fragments.

Based on the process outlined above, the compiler estimates iteration count for each processor and per-iteration cost. Then, by multiplying these two, it calculates the estimated workload for each processor. While this workload estimation may not be 100% accurate, it allows the compiler to rank processors according to their workloads and assign suitable voltage levels and frequencies to them as will be described in the next item. As an example consider the second loop nest shown above, parallelized using 4 processors. Assuming that our estimator estimates the cost of loop body as L instructions, the loads of processors $\mathtt{P}_0$, $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$ are $256 \times 1024 \times L$, $256 \times$ (1024-257+1) $\times L$, $256 \times$ (1024-513+1) $\times L$, and $256 \times$ (1024-769+1) $\times L$, respectively.

- **Voltage Assignment.** In this step, the compiler first orders the processors according to non-increasing workloads. After that, the highest voltage is assigned to the processor with the largest workload (the objective being not to affect the execution time to the greatest extent possible). Then, the processor with the second highest workload gets assigned to the *minimum voltage level* V_k *available* (where $1 \leq k \leq s$) that does not cause its execution time to exceed that of the processors with the largest workload. In this way, each processor gets the minimum voltage level (to save maximum amount of power) without increasing overall parallel execution time of the nest (which is determined by the processor with the largest workload). Continuing with the example above, suppose that we have two voltage/frequency levels (that is, V_1/f_1 and V_2/f_2, assuming $s = 2$ and $V_1/f_1 > V_2/f_2$), we first determine the execution time taken by processor $\mathtt{P}_0$ (denoted T_0). Then, for each other processor, we use V_2/f_2 if doing so does not cause their execution times to exceed T_0. If any of these execution times exceeds T_0 (when using V_2/f_2), we switch back to V_1/f_1 for that processor.

The success of our strategy critically depends on two important factors. First, there should be some load imbalance to exploit between different processors. This is because if there is no such imbalance then it is reasonable to execute each processor with the highest voltage/frequency. Second, the compiler-based workload estimation should be reasonably accurate. If this is not the case, then we may assign a

wrong voltage level/frequency to a processor, which may in turn impact overall execution time. In fact, in this scheme, the only time we pay some penalty is when our compiler-based workload estimation is not very accurate. In our experiments, we quantify this penalty in detail.

5. Additional optimizations

In this section, we discuss how the effectiveness of our strategy can be further increased using additional optimizations.

5.1. Inter-nest optimization

In the description of our strategy above, we assumed that the processors will synchronize at the end of each loop nest (before they start executing the next loop nest). As noted by Tseng [20], such a global synchronization presents two major problems. First, to implement such a synchronization, the compiler needs to generate extra (synchronization) code and insert it in the application code. Obviously, this code presents extra performance and power overhead at runtime. Second, since this synchronization requires all processors to wait for the slowest one, it makes poor use of available resources (from the performance angle). Consequently, allowing a processor to continue without waiting for the slower ones can allow small perturbations in processor execution times to even out, thereby improving overall performance (by taking advantage of the loosely-coupled nature of chip multiprocessors). However, determining when it is safe to allow a processor to continue without synchronization requires extra compiler analysis. In this study, we implemented a strategy that takes a number (called b) as a parameter, and for each loop nest, allows a processor to continue for at most b next nests if doing so does not violate any data dependences.

5.2. Voltage/frequency reuse

Another optimization can be performed by being more careful in voltage assignment. Up to this point in our discussion we assumed that the processor assignment for each loop nest is done independently of the other nests. As a result of this, as we move from one loop nest to another the same processor can get assigned different voltage levels. Consequently, we pay a penalty (in terms of both performance and energy consumption) for changing voltage levels. This penalty can be minimized by reusing the same voltage as much as possible for the same processor throughout the execution. This can be achieved as follows. Suppose that in loop nest i, we used voltage level V_k for processor j. When we move to loop nest $i+1$ if we need to assign voltage level V_k to a processor, we use processor j for that. This can be repeated for each neighboring loop nest pair, and in this way, the processors reuse their voltage levels as much as possible.

5.3. Adaptive parallelization

So far in our treatment of the subject, we have assumed that we use all available processors in execution of all nests in the application. However, it is known from prior research [9] that, in some cases using fewer processors (and shutting off the unused ones along with their L1 caches) can result in a better energy consumption behavior. We also conducted experiments with an adaptive strategy, where each loop nest is first profiled using different number of processors in conjunction with our optimization strategy. After the profiling, for each loop nest, we identified the ideal number of processors, and used it in the actual execution. It should be noted that in adaptive parallelization we use fewer number of processors than available (this means some performance loss); however, turning off unused processors along with their L1 caches can bring energy benefits.

Table 1. Base simulation parameters used in our experiments.

Parameter	Default Value
Number of Voltage/Frequency Levels	8
Lowest/Highest Voltage Levels	0.8V/1.4V
Frequency Step Size	30MHz
Voltage/Frequency Transition Penalty	10 cycles/2.10nJ
L1 Size	8KB
L1 Line Size	32 bytes
L1 Associativity	4-way
L1 Latency	1 cycle
L2 Size (Shared)	2MB
L2 Associativity	4-way
L2 Line Size	64 bytes
L2 Latency	10 cycles
Memory Access Latency	100 cycles
Bus Arbitration Delay	5 cycles
Replacement Policy	Strict LRU
L1 Energy (per access)	1.14nJ
L2 Energy (per access)	2.56nJ
Main Memory Energy (per access)	23.10nJ

5.4. Combining cyclic and block iteration allocations

As has been discussed earlier in the paper, one may also opt to use cyclic distribution of loop iterations across processors. Since our framework is able to estimate the number of cache misses, we can potentially have a better strategy as follows. For each loop nest, we can calculate the number of misses for both block and cyclic wise allocations and select the strategy that generates the best energy savings under a performance (execution cycles) constraint. We can refer to such a strategy as *hybrid* since it makes use of both block and cyclic wise allocation.

6. Experiments

We tested the effectiveness of our algorithm in reducing energy consumption of chip multiprocessor using six array-intensive programs: 3D, DFE, LU, SPLAT, MGRID, and WAVE5. 3D is an image-based modeling application that simplifies the task of building 3D models and scenes. DFE is a digital image filtering and enhancement code. LU is an LU decomposition program. SPLAT is a volume rendering application which is used in multi-resolution volume visualization through hierarchical wavelet splatting. Finally, MGRID and WAVE5 are C versions of two Spec95FP applications. These C programs are written in such a fashion that they can operate on inputs of different sizes. The default configuration parameters used in our experiments are given in Table 1, and these are the values that are used unless explicitly stated/varied in the sensitivity experiments.

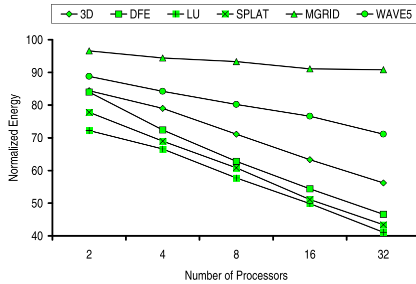

Figure 3. Normalized energy consumption with different number of processors (8 voltage levels).

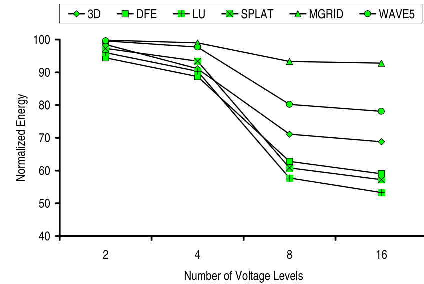

Figure 4. Normalized energy consumption with different voltage levels (8 processors).

To conduct our experiments, we modified Simics [18]. Simics is a full system simulation platform that can simulate both uniprocessor and multiprocessor machines. All energy results reported in this section include the energy spent in CPUs, their caches, and main memory and have been *normalized* with respect to the energy consumption when no voltage scaling is used and each processor is operated with maximum supply voltage and frequency.

The graph in Figure 3 gives the normalized energy consumptions with different number of processors. We can make two main observations from this graph. First, all our six applications get some energy benefit from our approach with all processor sizes experimented. Second, our energy savings get better with increased number of processors. This is because a larger number of processors means more load imbalance to optimize, and our approach takes advantage of it. When considering individual applications, one can see that MGRID and WAVE5 perform poorly as compared to the others, mainly because these applications have very few cases where our approach is applicable. In comparison, LU benefits much from increasing the number of processors since most of its few loops exhibit significant amount of load imbalance. Overall, the average savings across all six applications are between 16.03% (for the two processor case) and 41.80% (for the thirty-two processor case). To evaluate the impact of the number of voltage levels on energy savings, we also performed experiments with different number of voltage levels. The results are presented in Figure 4 for the 8 processor case. One can easily see from this graph that the number of voltage levels has a significant impact on energy behavior. In particular, the difference in going from 4 levels to 8 levels is dramatic; the corresponding savings are 6.63% and 29.02%. Increasing the number of voltage levels further (to 16) does not bring too much additional energy benefits since there is little scope left to be optimized (beyond what could be optimized using 8 levels). It should also be mentioned that when we have only 2 levels, the average saving across all applications is only 2.40%. This poor results is due to the fact that our strategy tries not to increase execution cycles as much as possible. Consequently, in many cases (when we have only 2 voltage levels) the compiler cannot use the lower voltage for a processor (even though the processor has low workload) since doing so would increase execution cycles dramatically.

7. Concluding remarks

A chip multiprocessor lowers the number of functional units per processor, and distributes separate tasks/threads to each processor. This paper has evaluated a compiler-directed strategy that allows different processors to use different voltage levels/frequencies to take advantage of the load imbalances stemming from loop parallelization. Our results with six applications clearly demonstrate the effectiveness of our strategy and makes a case for voltage-sensitive loop parallelization. Our results also show that it is possible to increase energy savings further by employing voltage/frequency reuse, adaptive parallelization, and internest optimization.

References

[1] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *Proceedings of International Symposium on Computer Architecture,* Vancouver, Canada, June 12–14 2000.

[2] S. Carr, K. S. McKinley, and C. Tseng. Compiler Optimizations for Improving Data Locality. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, October 1994.

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

[4] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. *Proceedings of the 11th ACM International Conference on Supercomputing,* July, 1997.

[5] K. Govil, E. Chan, and H. Wasserman. Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU. *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking,* November 1995.

[6] D. Grunwald, P. Levis, K. Farkas, C. Morrey III, and M. Neufeld. Policies for Dynamic Clock Scheduling. *Proceedings of the 4th Symposium on Operating System Design and Implementation,* October 2000.

[7] C.-H. Hsu and U. Kremer. Dynamic Voltage and Frequency Scaling for Scientific Applications. *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing,* August 2001.

[8] Intel XScale Technology. http://www.intel.com/design/intelxscale/.

[9] I. Kadayif, M. Kandemir, and U. Sezer. An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors. In Proc. *Design Automation Conference,* New Orleans, LA, June 2002.

[10] A. Klaiber. The Technology Behind Crusoe Processors. *Transmeta White Paper,* January 2000. http://www.transmeta.com/about/press/white_papers.html.

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

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

[13] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. *Proceedings of the 12th International Symposium on System Synthesis,* 1999.

[14] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Chip Multiprocessor. *Proceedings of the 7th Intl Conference on Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1996, pp. 2–11.

[15] POWER4 System Microarchitecture, *White Paper,* http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html

[16] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-Conscious Compilation Based on Voltage Scaling. *Proceedings of ACM SIGPLAN Joint Conference LCTES'02 and SCOPES'02,* Berlin , Germany, June, 2002.

[17] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. *Proceedings of the International Conference on Computer-Aided Design,* November 2000.

[18] SIMICS. http://www.virtutech.com/simics/simics.html.

[19] J. P. Singh and D. Culler. *Parallel Computer Architecture: A Hardware-Software Approach,* Morgan-Kaufmann, 1998.

[20] C.-W. Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. *Proceedings of 5th ACM Symposium on Principles and Practice of Parallel Programming,* Santa Barbara, CA, July 1995.

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. *Proceedings of the 1st Symposium on Operating Systems Design and Implementation,* November 1994.