Energy-Aware Computation Duplication for Improving Reliability in Embedded Chip Multiprocessors *

G. Chen, M. Kandemir, and F. Li
Computer Science and Engineering Department
Pennsylvania State University
e-mail: {guilchen,kandemir,feli}@cse.psu.edu

Abstract— Compilers designed for current embedded systems must be capable of addressing multiple constraints such as low power, high performance, small memory footprint and form factor, and high reliability at the same time. In particular, optimizing for one constraint should be performed carefully, considering its impact on other constraints. Recent trends indicate that transient errors are becoming increasingly important in embedded systems. Focusing on an embedded chip multiprocessor and array-intensive applications, this paper demonstrates how reliability against transient errors can be improved without impacting execution time by utilizing idle processors for duplicating some of the computations of the active processors. It also shows how a balance between power savings and reliability improvement can be struck using a metric called the energy-delay-fallibility product. Our experimental results indicate that the "percentage of duplicated computations" is a useful high-level metric for studying the tradeoffs among performance, power, and reliability.

I. Introduction

Today's embedded systems are expected to satisfy multiple, and often conflicting, criteria (constraints) such as low power, high performance, small memory footprint and form factor, and high reliability. Therefore, the entire system design cycle employed in the past, which includes software and hardware, must be re-thought based on this multi-criteria requirement. For example, most of the software compilation techniques developed over the years target mainly at optimizing performance. Recently, compiler designers have also focused on memory footprint estimation/reduction [5,20–22] and power/energy optimizations [9, 14]. However, recent increase in transient error rates due to scaling technology and employment of low power techniques made it imperative to consider reliability as a first class optimization metric. The important point though is the reliability measures must be well-balanced against performance and power concerns. In other words, while ensuring reliable execution as much as possible, one also needs to be careful in not increasing execution cycles or power consumption excessively.

One way of addressing growing complexity problem of embedded system designs is chip multiprocessors [2, 3, 13]. The idea is to accommodate multiple simple cores within the same die instead of a complex single core based architecture. Prior research [13, 15] has already discussed several advantages of chip multiprocessors over complex single processor based solutions, which include appropriateness to high-level code parallelization and design and verification advantages. Performance/verification related issues have been addressed in studies like [16], whereas studies such as [18] and [10] focused on the problem of reducing power consumption. An important observation made by one of the prior studies [10] is that, in executing array-intensive codes in a chip multiprocessor, not all the cores are used all the time. That is, at any given time, certain number of processors are idle and can potentially be switched off or put in a low-power operating mode to save energy.

In this work, we explore an alternate use of such idle processors in an embedded chip multiprocessor. Specifically, focusing on embedded array-intensive applications, we demonstrate that idle processors can be used for *duplicating* some of the computations of the active processors, thereby improving overall reliability against transient errors. In this approach, a computation can be duplicated once to detect transient errors (our focus in this paper) or twice to detect and correct them. It should also be noticed that not all of the idle processors need to be used for duplicating computation. In fact, some of them can still be placed into a low-power mode to save power, which is a particularly promising approach in an environment where both reliability and power consumption are important metrics to consider. Whether an idle processor should be used for saving power or increasing reliability depends on the relative importances of reliability and power as well as allowable power consumption and acceptable error levels.

It is to be emphasized that reliability concerns due to transient errors are becoming an increasingly pressing problem for embedded systems. This is mainly because of two reasons. First, as devices are being pushed into very deep sub-micron technologies (< 250 nano-meter), reliability is becoming an important issue. Some of the growing effects are the so-called "transient errors", which are due to temporary conditions of usage characteristics and the environment. Cross-coupling, ground bounce, external terrestrial radiations create more and more unpredictable transient and soft errors which affect system reliability. Transient errors caused due to radiations especially have been studied very closely in industry. The second reason is that many embedded systems employ several mechanisms that scale voltage or place unused components into low-power modes (sleep states), with the objective of reducing energy consumption. This in turn increases the vulnerability of these systems to transient errors. A recent study [4] investigates the relationship between power-saving strategies and cir-

*This work is supported in part by NSF Career Award #0093082 and by a grant from GSRC.

0-7803-9451-8/06/$20.00

cuit reliability.

Since we want to measure the impact of our approach on performance, power, and reliability, previous evaluation metrics such as execution cycles, performability, or energy-delay product are not very appropriate for our purposes. Instead, we employ a different metric called the *energy-delay-fallibility* product (EDF), where fallibility in this context is the opposite of reliability. Using this metric, we study how the different divisions (partitionings) of the idle processors between those that execute duplicated computations and those that are placed into a low-power mode to save power can affect the value of the energy-delay-fallibility product. We also study the impact of the duplication granularity for adaptation (i.e., application based versus loop nest based) on the value of the energy-delay-fallibility metric. Note that, using new metrics for characterizing system behavior in terms of energy efficiency, reliability, computation performance and battery lifetime has been a popular research topic recently [19].

We automated our reliability-oriented approach within a parallelizing compiler and tested it using seven array-intensive benchmark codes. Our results, collected using the Simics simulation toolset [17], demonstrate that not just the different applications require different percentages of idle processors to be used for computation duplication, but also even within the same application, the different loop nests work best (from the perspective of the energy-delay-fallibility product) with the different percentages of idle processors being used for duplication. In other words, the amount of duplication should be tuned at a loop nest granularity. While more computation duplication intuitively means better reliability, the latter concept depends also on the frequency of transient errors and the patterns these errors exhibit. The results indicate that the "percentage of computations duplicated" is a reasonable indicator for reliability. In this paper, as long as there is no confusion, we use the terms "percentage of computations duplicated" and "percentage of processors used for duplicating computations" interchangeably.

The remainder of this paper is structured as follows. The next section summarizes the architectural abstraction presented to our approach. Section III discusses the details of our computation duplication strategy. Section IV introduces our experimental setup and reports the experimental results from our implementation. This experimental evaluation considers both the error-free case and the cases with errors. Section V concludes with a summary of our major observations.

II. Chip Multiprocessor Architecture and Execution Model

We focus on an embedded chip multiprocessor of the shared memory type. In this architecture, multiple processor cores (typically between 4 and 32) reside on the same chip. Each core has private L1 instruction and data caches and there is also a large unified L2 cache shared by all the cores. We also assume existence of an off-chip memory, whose access latency and power consumption are typically much larger than the corresponding values for the on-chip L1 and L2 caches. In this architecture, data sharing and communication among processors is achieved using shared memory components (i.e., through L2 cache and off-chip memory). Note that, several chip multiprocessor proposals [13, 15] from academia and industry fit in this abstraction.

We focus on execution of array-intensive applications on this chip multiprocessor. It is to be emphasized that, array-intensive applications frequently appear in many embedded domains where reliability is also a concern. In this work, an array-intensive embedded application is parallelized by considering each loop nest in turn, and distributing its iterations across processors. How exactly the loops are parallelized is orthogonal to the focus of this work. For this purpose, one can employ either user-assisted methods or automatic compiler support. For a given loop nest, the set of loop iterations assigned to a particular processor is called its *local iteration set* or *local iteration space* for that processor with respect to that nest. Our unit of duplication (for reliability purposes) in this paper is a local iteration space. That is, we duplicate the local iteration space of an active processor on an otherwise idle processor. When a processor core is switched to a low-power mode, its L1 cache is also assumed to be put in the low-power mode. While in the low-power mode, a processor/L1 pair consumes a fraction of the energy that they would consume in the full active mode, and the L1 cache maintains its contents. Neither L2 cache nor off-chip memory is turned off during program execution.

III. Computation Duplication

Figure 1 illustrates our approach to computation duplication. In Figure 1(a), a loop nest is parallelized (either automatically by a parallelizing compiler or through user help), and m out of total n processors (P_1 through P_m) are used to execute the loop nest. In a previous work [10], all the idle processors (P_{m+1} through P_n) along with their L1 caches are placed into a low-power mode to save energy. In our work, on the other hand, we utilize some of these idle processors to improve reliability by executing on them duplicates of the computation performed by the active processors (P_1 through P_m). What we mean by computation in this context is the local iteration space assigned to an active processor as a result of code parallelization. In Figure 1(b), the local iteration spaces of r processors (P_1 through P_r) are duplicated and executed on processors P_{m+1} through P_{m+r} to improve the reliability of the computation. When there is no confusion, we say that P_i ($1 \leq i \leq r$) is a *primary* processor, and P_{m+i} is the *duplicate* of P_i. It is to be noted that, these concepts of primary and duplicate processors are relative to a given loop nest; i.e., a given processor can be primary in one nest and duplicate in another.

To improve reliability of execution for a given primary processor, we have to check its execution with that of its duplicate to see whether their results agree. A straightforward way of achieving this would be letting the primary and its duplicate run in a lock-step fashion, during which they compare their results after executing each and every statement. Although such a lock-step execution can be implemented rather easily, it can also generate a lot of communication and synchronization activities between the primaries and their duplicates, and the overheads it incurs in terms of both performance and energy consumption can be intolerable for an embedded system. This is particularly true considering the fact that communication and synchronization costs can be critical in a chip multi-

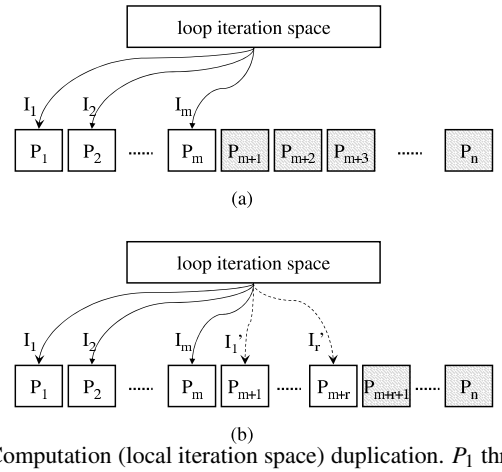


Fig. 1. Computation (local iteration space) duplication. P_1 through P_n represent the processors in the chip multiprocessor. The idle processors are shaded. (a) The loop nest is parallelized and set to execute on m processors; I_i ($1 \leq i \leq m$) is the set of iterations assigned to P_i as a result of parallelization, i.e., its local iteration set. (b) I_1 to I_r (where $r \leq m$) are assumed to be duplicated and the duplicated computations are assigned to P_{m+1} through P_r. Dashed curves represent the duplicated iteration sets. I_i' is the duplicate of I_i.

$$A[i] = C[i+1] * D[i] + E[i];$$
$$B[i] = C[i-1] - D[i-1];$$
(a)

$$A[i] = C[i+1] * D[i] + E[i];$$
$$CHECK[prid] += A[i];$$
$$B[i] = C[i-1] - D[i-1];$$
$$CHECK[prid] += B[i];$$
(b)

$$CHECK[prid] += C[i+1] * D[i] + E[i];$$
$$CHECK[prid] += C[i-1] - D[i-1];$$
(c)

Fig. 2. Adding checksum to a code segment. *prid* is the id of a given processor, and $CHECK[prid]$ is the corresponding checksum. (a) Original code (loop body). (b) Transformed code to be executed by the primary processor. (c) Transformed code to be executed by the duplicate processor.

$$A[i] = B[i] + C[i];$$
$$B[i] = C[i] - 10;$$
(a)

S_{b1}: $A[i] = B[i] + C[i];$
S_{b2}: $CHECK[prid] += A[i];$
S_{b3}: $B[i] = C[i] - 10;$
S_{b4}: $CHECK[prid] += B[i];$
(b)

S_{c1}: $CHECK[prid] += B[i] + C[i];$
S_{c2}: $CHECK[prid] += C[i] - 10;$
(c)

$$CHECK[prid] += B'[i] + C[i];$$
$$B'[i] = C[i] - 10;$$
$$CHECK[prid] += B'[i];$$
(d)

Fig. 3. An example code segment that illustrates the need for array duplication. (a) Original code. (b) Transformed code to be executed by the primary processor. (c) Incorrect version of the transformed code to be executed by the duplicate processor. (d) Correct version of the transformed code with array duplication to be executed by the duplicate processor.

processor. Instead, in our work, we use a more efficient way of comparing these two executions, which is as follows. We associate a *checksum* with each processor (i.e., with both primary and its duplicate), and all these checksums are initialized to zero at the beginning. Each time a statement finishes its execution and produces a result, we add it to the corresponding checksum. After all the loop iterations complete, we compare the primary's checksum with its duplicate's checksum to see whether they are equal. Note that, in this approach, we do not need any output (except for the checksum) from the duplicate processor. Therefore, in most cases, in the duplicate, all the results of computations can be directly fed to the checksum without being written into any array or variable, as would be the case in the original code. This also helps reduce the energy overhead associated with the duplicate significantly. Figure 2 gives an example of how a code segment can be transformed to generate checksum. Figure 2(a) is the original code in the loop body (assuming that i is the loop index variable); Figure 2(b) is the transformed code to be executed by the primary; and Figure 2(c) is the transformed code to be executed by the duplicate. It can be seen that the code to be executed by the duplicate does not update arrays A and B due to the reason explained above.

The transformation used in Figure 2 may not work correctly if an array element is both read and written in the iteration space (i.e., when there exists a data dependence involving the array in question). Figure 3 presents such an example. In statement S_{b1}, we use the old value of $B[i]$. To be consistent, S_{c1} should also use the old value of $B[i]$. Otherwise, the checksums computed in Figure 3(b) and Figure 3(c) will be different from each other, even in the absence of any transient error, because the input values used to calculate them are different. Therefore, S_{c1} should be executed before S_{b3} to ensure that the old value of $B[i]$ is not overwritten by S_{b3}. However, doing so means that we need to synchronize the primary processor and its duplicate at the statement level, which is what we want to avoid by introducing checksums at the first place. Our approach to address this problem is presented in Figure 3(d). In this solution, array B is duplicated, and B' represents the duplicate. We replace B with B' in the code depicted in Figure 3(c), and we obtain the code in Figure 3(d). Notice that, although not shown here explicitly, we need to initialize B' with B's old values before the loop is entered since the old values of B will be used before they are written (updated) within the loop. Using array duplication, the synchronization problem discussed above due to data dependences is solved. It should be observed though that such data duplications bring extra overheads in terms of memory space occupation, performance (execution cycles), and energy consumption. However, our experience with numerous application codes shows that such cases (that require data duplication) do not happen very frequently. In fact, we observed during our experimental evaluation that, in only six out of the twenty four loops in our study need array duplication, and *we take all the associated overheads into account in our experimental evaluation.* The code transformations illustrated in Figures 2 and 3 have been automated within our compiler.

It should be noticed that this computation duplication catches all types of transient errors during loop execution (*as long as either the primary and the duplicate executes correctly*), *not just the memory related errors.* That is, it also captures the CPU errors that might occur during the execution of the duplicated loops. Therefore, it is very general. While it is possible that it can fail in cases where a wrong computation still generates a correct checksum, such cases are expected to be very rare in practice. The presented scheme is different from the prior code duplication related studies such as [1,6,8], as we focus on an embedded chip multiprocessor and try to minimize a different metric, which captures performance, energy, and reliability. However, there is still a chance that a transient error can strike during execution of an unduplicated computation, which cannot be detected by our approach. In other words,

TABLE I
MAJOR SIMULATION PARAMETERS AND THEIR VALUES.

Parameter	Value
Number of Processors	8
L1 Instruction Cache	8KB 2-way associative 32 byte blocks
L1 Data Cache	8KB 2-way associative 32 byte blocks
L2 Cache	1MB 4-way associative 64 byte blocks
L1 Dynamic Energy Consumption	0.16 nJ/access
L2 Dynamic Energy Consumption	0.65 nJ/access
r	0.1
Reactivation Latency	4 cycles
Off-Chip Memory Energy	6.32 nJ/access
Off-Chip Memory Access Latency	80 cycles
On-Chip Bus Arbitration Delay	5 cycles
Replacement Policy	Strict LRU

while we increase the resilience of execution against soft errors, the number of errors we actually detect is another matter, and depends on error injection rate, error injection pattern, and other factors.

IV. EXPERIMENTS

A. Setup

We used a chip multiprocessor simulator for our experiments, built upon Simics [17]. Simics is a platform for full system simulation that can run actual firmware, complete kernel, and driver codes. It is sufficiently abstract to achieve good performance levels, and it provides both functional accuracy for running commercial workloads and sufficient timing accuracy to interface to detailed hardware models. In particular, it allows us to model the overheads incurred by our approach accurately. Our simulator keeps track of the number of instructions executed by each processor and data/instruction accesses to different memory components (L1, L2, and off-chip memory). We also embedded energy models into this simulator. These energy models are access based, and compute energy of a component by multiplying the number of accesses to that component with a fixed (component specific) per access energy consumption. The per access energy consumption value for each component is obtained through profiling. The necessary code modifications to insert checksum computations in the source codes are automated within a parallelizing compilation framework [7]. The increase in compilation time caused by our approach was about 19% when averaged over all the codes used in our evaluation. The largest compilation time increase was approximately 41%.

Table I gives the major simulation parameters used in this study. The parameter r specifies the magnitude of the leakage power consumption when a processor core/L1 cache is placed into the low-power mode. More specifically, when r=r*, this means that the leakage energy consumed by the core/L1 pair in the low power mode is r*x100% of the leakage energy consumption of an active processor core/L1 pair. We assume that the leakage energy per cycle for an 8KB SRAM (our L1) is equal to the dynamic energy consumed per access to a 32 byte data from that SRAM, similar to the assumption made by [11].

The seven benchmark codes used in this study and their important characteristics are given in Table II. These benchmark codes are extracted from the Perfect Club, Spec, Livermore, and DSPStone benchmark suites. The values listed in this table are obtained by executing the benchmarks in our simulation environment *without* any power management and *without* any computation duplication. All benchmark codes have been run to completion. In the rest of our discussion, we refer to this version of a benchmark as the *base version,* or the *original version.* The third column gives the number of cycles. The last two columns give the dynamic and leakage energy consumptions under the 70nm process technology. These values include the energies consumed in the processor cores, L1 and L2 caches, and off-chip main memory.

TABLE II
BENCHMARK CODES USED IN OUR EXPERIMENTS AND THE STATISTICS COLLECTED WHEN NO POWER OPTIMIZATION AND NO DUPLICATION IS USED.

Benchmark Name	Number of Nests	Cycles (Million)	Dynamic Energy (uJ)	Leakage Energy (uJ)
3step-log	3	14688	20195.27	29390.57
adi	2	217	589.45	706.24
btrix	7	82336	92524.07	83096.42
eflux	2	1236	2018.51	2496.56
full-search	3	97640	137887.96	18937.01
n-real-updates	3	160	395.45	423.66
tsf	4	246	752.84	282.55

As stated earlier, our approach can work under any loop parallelization strategy. The particular strategy used in this work is based on locality of reference. In this strategy, a loop nest is parallelized such that each processor accesses data mostly with temporal or spatial reuse. However, each loop is parallelized using the minimum number of processors; that is, increasing the number of processors beyond this minimum number does *not* improve performance any further. Table III gives statistics on the execution of the base version under the default machine configuration. Each column (starting with the second one) in this table corresponds to a loop nest in the application and each cell shows the minimum processor count (as explained above) that gives the best performance for that nest. That is, using more processors for the nest does not improve its performance further. An observation that one can make from the values in this table is that, in almost all the loop nests, there exist some idle processors, which can potentially be placed into the low-power mode to save energy, or can be used for duplicating some computation to increase error detection capabilities. In fact, most of the nests use only 4 or fewer processor (of a total of 8 processors) to generate the best execution cycles. We need to point out that this low processor utilization is typical in parallel processing (i.e., it is not a particular characteristic of the applications used in this study), and can be explained as follows. As we increase the number of processors over which a loop nest is parallelized, at least two types of overheads increase. The first of these is the time/energy spent in spawning the additional threads and finalizing them when the parallel loop execution is over. The second one is the synchronization costs due to increased inter-processor communication. In addition, some data dependences across loop iterations generate the best results with a particular processor count. All these factors are effective in preventing us from using all processors in the chip multiprocessor. As a result, going beyond a given processor size increases only the overheads without bringing any performance benefits. In particular, using more processors than necessary can have devastating results from the energy consumption angle.

TABLE III
MINIMUM NUMBER OF PROCESSORS THAT GENERATE THE OPTIMUM PERFORMANCE FOR EACH NEST OF EACH BENCHMARK CODE. NOTE THAT, EACH APPLICATION HAS A DIFFERENT NUMBER OF NESTS. NI REPRESENTS THE ITH NEST IN THE CORRESPONDING APPLICATION.

Benchmark	N1	N2	N3	N4	N5	N6	N7
3step-log	1	1	5				
adi	4	5					
btrix	2	1	7	6	1	3	8
eflux	2	3					
full-search	2	2	6				
n-real-updates	4	4	4				
tsf	1	7	2	4			

As stated earlier, we use the energy-delay-fallibility product (denoted EDF henceforth) as our metric in this paper. In our context, energy corresponds to the sum of the energies consumed in the processor cores, L1 and L2 caches, and off-chip memory, and delay is the parallel execution time (measured in cycles). Fallibility is the opposite of reliability; the latter being defined as the percentage of loop iterations that are duplicated. Clearly, we want the value of the EDF metric to be as small as possible. As mentioned earlier, our approach can involve array duplication in certain cases, and this can in turn affect the energy consumption and execution cycles of the application (due to the degradation in cache performance). *All these overheads are included in the EDF values presented in the next subsection.* While, as mentioned earlier, we can duplicate a local iteration space twice (instead of just once) to correct errors (e.g., through a majority voting based scheme), in this paper we present the results with single duplication only; i.e., we focus primarily on the error detection problem. It should be mentioned that the impact of our approach on original execution cycles is not excessive. In fact, there are only three potential reasons why the performance can be affected because of our approach. The first of these is due to array duplication. As we mentioned earlier, it does not occur very frequently, and as a result, the overheads it incurs are not excessive. The second overhead is due to comparing the checksums (which also involves inter-processor synchronization), and its impact is not very high. The third one is the reactivation cost when a processor/L1 pair is powered down, which is not very frequent (as it is done only between nest boundaries). Overall, we observed that the performance degradation due to our approach was less than 2% when averaged over all the benchmark codes in our experimental suite. In any case, the EDF values presented below include this small degradation in performance as well. However, before presenting the benefits of our approach, we give in Figure 4 the percentage contribution of each type of overhead. We see that, while the overheads due to array duplication dominate the others, the other two factors also play a role. However, as mentioned above, their cumulative impact on the performance is less than 2% on the average.

B. Results

Our first set of EDF results are presented in Figure 5. Each point on the x-axis represents a percentage of idle processors used for computation duplication. The remaining processors are placed into the low-power mode along with their L1 caches. Note that, these percentages are valid for each nest of each application. That is, in each loop nest, the same percentage of

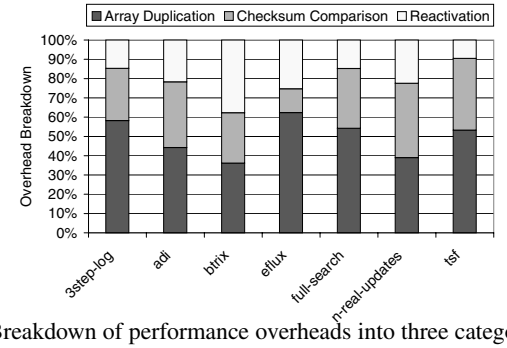


Fig. 4. Breakdown of performance overheads into three categories.

TABLE IV
STATISTICS WHEN 20% OF THE IDLE PROCESSORS ARE USED FOR COMPUTATION DUPLICATION. AS COMPARED TO TABLE III, WE HAVE MORE CYCLES AND MORE ENERGY CONSUMPTION HERE DUE TO DUPLICATION.

Benchmark Name	Cycles (Million)	Dynamic Energy (uJ)	Leakage Energy (uJ)
3step-log	15623	25253.15	35313.51
adi	242	756.49	836.72
btrix	91089	127330.08	104521.98
eflux	1318	2816.13	2753.38
full-search	103233	165538.25	25913.14
n-real-updates	182	520.32	519.68
tsf	276	969.60	387.67

idle processors are used for duplication. In the results shown in Figure 5, the EDF values for each application are *normalized* with respect to the EDF values when 20% of the idle processors are used for computation duplication (the absolute cycles, dynamic and static energy consumption values for the case when 20% of the idle processors are used for duplication are given in Table IV). It can be observed from the results in Figure 5 that the trends in general are similar across the different applications. Most of the applications have a decrease at first in EDF as we increase the percentage of idle processors used for computation duplication, which can be attributed to the increased reliability we have as more iterations are duplicated. However, we also observe that, beyond a certain point, using more idle processors for duplication increases EDF since the increased energy consumption and execution time (to a lesser extent) starts to offset the benefits brought by more duplicated iterations. Therefore, *it is important to pick a suitable percentage of duplication for a given application to reach a good tradeoff point between performance, energy, and reliability.* Only two applications, namely full-search and tsf, exhibit slightly different trends. Their curves keep decreasing as the percentage of duplicates increases. This can be attributed to the relative lower leakage energy consumed by these two applications (as compared to their dynamic energy consumptions). Lower leakage energy consumption means less energy savings brought by putting the idle processors into the low-power operating mode. Therefore, as we use more processors for duplication, the benefits coming from the increased reliability (i.e., the decreased fallibility) can be offset by the increased energy consumption (as far as the EDF metric is concerned). We can also observe from Figure 5 that the different applications reach their optimum (minimum) EDF values at different points. For example, eflux reaches its optimum result at 30%, whereas 3-step-log achieves its optimum result at 50%. As has been discussed

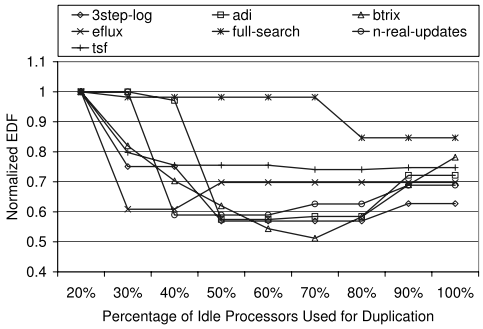
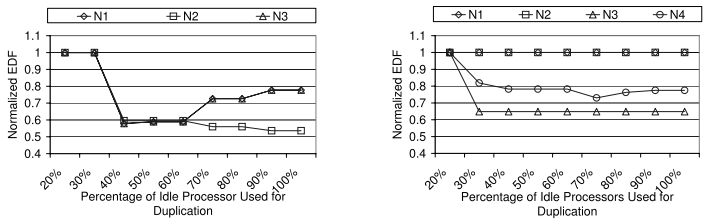


Fig. 5. The EDF values with the different percentage of idle processors being used for computation duplication. The EDF values for each application are normalized with respect to the EDF value when 20% of the idle processors are used for duplication.



(a) n-real-updates. (b) tsf.

Fig. 6. Normalized EDF for the different loop nests in n-real-updates (left) and tsf (right). n-real-updates has three loop nests, denoted by N1, N2, N3, and tsf has four loop nests, denoted by N1, N2, N3, and N4.

above, leakage energy consumption behavior can affect the optimum point. In addition to this, the overheads incurred by computation duplication are also a factor for some applications. If the overhead brought by computation duplication is high, a small percentage in duplication will be favored. Otherwise, using more processors for duplication can be expected to be more beneficial as far as the EDF metric is concerned. Overall, these results suggest that the percentage of computation duplication used should be tuned for each application separately.

Figure 6 presents the EDF curves for individual loop nests in two benchmarks: n-real-updates and tsf. It can be seen from Figure 6(a) that the curves for N1 and N3 in n-real-updates are almost overlapped, whereas N2 exhibits a different pattern. Specifically, one would prefer to use a relatively small percentage value, say about 40%, for N1 and N3, but use a large percentage value for N2, as suggested by the trend exhibited by the N2's curve. The curves for loop nests of tsf, shown in Figure 6(b), are relatively flat. It can still be observed though that the different loop nests have different trends. For example, N4 reaches its optimum point at 70%, while the other three loop nests are not very sensitive to the percentage change. All of these observations indicate that, to obtain the best tradeoffs, one should consider adaptive computation duplication in the granularity of loop nests, in addition to adaptive computation duplication at application level. To sum up, the results presented in Figure 5 and Figure 6 motivate for application-level and loop nest-level adaptation, respectively, in utilizing the idle processors.

V. Conclusions

Single metric based compilation strategies are not sufficient for current complex embedded systems, where multiple constraints (e.g., power, performance, reliability) are important and need to be accounted for at the same time. In this context, maybe the most useful class of optimizations are those that optimize for one metric without impacting the others excessively. Motivated by this view, in this paper we evaluate a reliability oriented compilation strategy for embedded chip multiprocessors based on computation duplication. Using a new metric, called the energy-delay-fallibility product (EDF), we study the impact of the percentage of idle processors used for computation duplication. Our results suggest that an adaptive scheme in choosing the number of idle processors for computation duplication is needed to achieve the best tradeoffs between performance, energy efficiency, and reliability.

References

[1] C. Bolchini. A Software Methodology for Detecting Hardware Faults in VLIW Datapaths. *IEEE Transactions on Reliability*, 52(4):458-468, December 2003.

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

[3] Chip Multiprocessing. ITWorld.Com, http://www.itworld.com/Comp/ 1092/CW-STO54343/

[4] V. Degalahal, R. Rajaram, N. Vijaykrishanan, Y. Xie , and M. J Irwin. The Effect of Threshold Voltages on Soft Error Rate. In *Proc. ISQED,* San Jose, CA, 2004.

[5] A. Fraboulet, K. Kodary, and A. Mignotte. Loop Fusion for Memory Space Optimization. In *Proc. the International Symposium on System Synthesis,* Montreal, Canada, September 30-October 3, 2001.

[6] C. Gong, R. Melhem and R. Gupta. Compiler-Assisted Fault Detection for Distributed Memory Systems. In *Proc. the Scalable High Performance Computing Conference*, Knoxville, TN, 1994.

[7] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, Dec 1996.

[8] J. G. Holm and P. Banerjee. Low Cost Concurrent Error Detection in a VLIW Architecture Using Replicated Instructions. In *Proc. the International Conference on Parallel Processing*, pp. 192-195, 1992.

[9] C-H. Hsu and U. Kremer. Single Region vs. Multiple Regions: A Comparison of Different Compiler-Directed Dynamic Voltage Scheduling Approaches. In *Proc. PACS Workshop,* Cambridge, MA, February 2002.

[10] I. Kadayif, M. Kandemir, and M. Karakoy. An Energy Saving Strategy Based on Adaptive Loop Parallelization. In *Proc. Design Automation Conference*, June 10-14, 2002, New Orleans, Louisiana, USA.

[11] S. Kaxiras, Z. Hu, M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. the 28th International Symposium on Computer Architecture,* Sweden, June 2001.

[12] S. Kim and A. Somani. Area Efficient Architectures for Information Integrity Checking in Cache Memories. In *Proc. ISCA,* May 1999.

[13] V. Krishnan and J. Torrellas. A Chip Multiprocessor Architecture with Speculative Multi-threading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture,* September 1999.

[14] M. Lorenz, L. Wehmeyer, and T. Drager. Energy-Aware Compilation for DSPs with SIMD Instructions. In *Proc. LCTES,* Berlin, Germany, 2002.

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Chip Multiprocessor. In *Proc. ASPLOS,* 1996, pp. 2–11.

[16] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer,* (Vol. 36, No. 4), April 2003.

[17] Simics Tool-set. http://www.simics.com.

[18] R. Sasanka et al. The Energy Efficiency of CMP vs. SMT for Multimedia Workloads. In *Proc. ICS,* June 2004.

[19] P. Stanley-Marbell and D. Marculescu. Dynamic Fault-Tolerance and Metrics for Battery Powered, Failure-Prone Systems. In *Proc. ICCAD,* San Jose, CA, 2003.

[20] M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-Independent Storage Mapping in Loops. In *Proc. the International Conference on Architectural Support for Programming Languages and Operating Systems,* October, 1998.

[21] W. Thies et al. A Unified Framework for Schedule and Storage Optimization. In *Proc. PLDI,* Snowbird, UT, June, 2001.

[22] Y. Zhao and S. Malik. Exact Memory Size Estimation for Array Computations without Loop Unrolling. In *Proc. DAC,* June 1999.