Dynamic Compilation for Energy Adaptation [*]

P. Unnikrishnan, G. Chen, M. Kandemir
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{unnikris,gchen,kandemir}@cse.psu.edu

D. R. Mudgett
School of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA
drm1@psu.edu

ABSTRACT

While previous compiler research indicates that significant improvements in energy efficiency may be possible if properly optimized code is used, the energy constraints under which a given application code should be optimized may not always be available at compile-time. More importantly, these constraints may change dynamically during the course of execution. In this work, we present a dynamic recompilation/linking framework using which the energy behavior of a given application can be optimized while the application is being executed. Our preliminary experiments indicate that large energy gains are possible through dynamic code recompilation/linking at the expense of a relatively small increase in execution time.

1. Introduction and Problem Motivation

The results from previous compiler research (e.g., [2]) indicate that significant improvements in energy efficiency may be possible if properly optimized code is used; however, in some cases, there may be inherent conflicts between optimizations for energy efficiency and those designed purely for maximum execution-time speedup. Furthermore, the information needed to make a decision between these two goals is not necessarily known a-priori at compile time. To the contrary, such information is likely to depend on the changing system operating environment, available only by feedback of operating conditions at run-time. Hence, it appears that dynamic recompilation of an executing application, in response to a run-time status signal, may be a useful way to achieve a reasonable balance between different goals such as execution time and energy efficiency.

Fundamentally, any dynamic recompilation and binary modification system for energy optimization should answer the following questions:

- In practice, how can a given target code be dynamically recompiled or directly loaded (linked) from precompiled modules in an efficient, safe, and reasonably transparent way?
- What is a reasonable tradeoff between the fixed overhead involved in dynamically recompiling/linking code segments versus the variable cost of continuing to run suboptimal code when energy constraints change?
- When is it better to delay recompilation of code until needed (i.e., lazy recompilation, with its attendant high reconfiguration overhead but lower memory requirement) versus simply loading multiple, precompiled, cached codes?
- What is the potential energy/performance gain achievable through such a reconfigurable, dynamic software environment?

The first question is a practical issue of what software is available or can be developed to instrument code, that is dynamically on-the-fly replace old code with new code, without recompiling the entire application. Fortunately, this problem has already been studied to some extent and some software infrastructure is available for use. The answers to the second, third, and fourth questions, on the other hand, depend heavily on the situation; however, we believe that there are two important principles that might guide us:

- Changes to the code of a running application should not be made unless one is sure that they will payoff. In other words, unless one is reasonably confident that an energy level will be maintained for a sufficiently long period that the overhead of dynamic recompilation/linking will be amortized by the longer-term energy benefits, no change should be made.
- The optimizations made to the application code should be compatible with the current energy constraints. In cases where the potential runtime energy constraints can be predicted at compile-time and there is sufficient instruction memory, multiple versions of the same code can be precompiled and the most suitable version can be activated at runtime. In cases where dynamic constraints cannot be predicted or a memory space problem exists, on the other hand, the code should be recompiled at runtime.

This paper is primarily concerned with the implementation of a dynamic recompilation/linking system for runtime energy optimization. We focus on an operating environment where energy constraints change from time to time. Examples of these changes are running low in battery, a thermal emergency signal, and battery being recharged. We demonstrate how our dynamic recompilation/linking framework reacts to these runtime variations in energy constraints and show that the cost of dynamic recompilation is not very high. While there has been some recent work on compilation for reducing energy consumption, to the best of our knowledge, this is the first study that investigates the use of dynamic recompilation/linking for energy reduction. Our work is a first step in investigating the suitability of dynamic code compilation for energy. Consequently, it focuses on some of the important issues only and postpones complete treatment of individual problems to future studies.

In the last eight years or so, there has been significant interest in dynamic recompilation and on-the-fly executable reconfiguration. Groups at IBM Watson Research Center [5], MIT/University of Arizona [6], University of Washington [8] as well as two different groups at the University of Wisconsin [10, 12] have developed infrastructures for dynamic recompilation. All these projects, however, aim at reducing execution time of the application at hand rather than improving its energy behavior.

The rest of the paper is organized as follows. In Section 2, we introduce our dynamic recompilation/linking environment for energy and describe its implementation. In Section 3, we present experimental data showing the effectiveness of our approach. Finally, in Section 4, we present our conclusions.

[*]This work is supported by the NSF Awards 0093082 and 0103583, and by an award from the PDG.

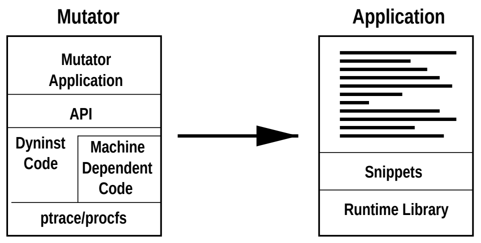

Figure 1: Abstractions used in the Dyninst API.

2. Dynamic Recompilation/Linking for Energy

2.1 Dyninst Software

To implement our dynamic compilation infrastructure for energy, we examined several binary reconfiguration tools such as DAISY [5], VCODE [6], DyC [8], EEL [10]. We decided not to use any of these tools due to reasons such as platform limitations, complex low-level implementations, license problems, or insufficient features supported. The Dyninst software from the University of Maryland [1] was chosen for our study. Dyninst is a post-compiler program manipulation tool which provides an Application Program Interface (API) called DyninstAPI for program instrumentation. Dyninst also provides Dyner, a TCL-based interactive command-line tool based on DyninstAPI library. The Dyner command set implements the functionality of the DyninstAPI. In this work, we chose to use the C++ interface of DyninstAPI directly rather that using the TCL-based Dyner. This choice gives our implementation maximum flexibility. This also reduces the overhead incurred due to the intermediate TCL layer.

Using the DyninstAPI library, it is possible to instrument and modify application programs during execution (i.e., as they are running). DyninstAPI is itself a C++ class library which can be included and directly called from a C++ program. This API is based on the idea of dynamic instrumentation technology developed as part of the Paradyn Parallel Performance Tools project [12] at the University of Wisconsin. A key feature of the DyninstAPI interface is that it permits insertions and alterations in a running program unlike other post-compiler instrumentation tools such as EEL [10] or ATOM [13] that permit code to be inserted into the binary before it starts to execute. Using this API, a program can create a new piece of code and insert it into another program while the latter is executing. The program being modified is able to continue execution and does not need to be recompiled entirely. This is ideal for the operating environment considered in this paper where dynamic changes in energy constraints enforce modifying parts of a running application. There has been previous work involving Dyninst for performance steering of programs. In contrast to those studies, in this work, we employ Dyninst for energy optimization.

The overall structure of the API and its implementation are shown in Figure 1 (taken from [1]). There are two processes, called the *mutator* and the *application* (or *mutatee*). The left side of the figure shows the code for the mutator process that contains calls into the Dyninst API. It also contains the code that implements the runtime compiler and the utility routines to manipulate the application process (shown below the rectangle labeled API) as well as profiling/tracing tools. The right half of the figure shows the application process with the original code of the program shown in the top part of the figure. The bottom two parts of the application are the snippets that are inserted into the program, and the runtime library that supports the Dyninst API. To perform our experiments, we installed Dyninst on Sun Solaris. All experiments reported in this paper are performed on this platform, using a Sun Sparc-based machine running at 400 MHz with 512 MB main memory and 2 MB cache.

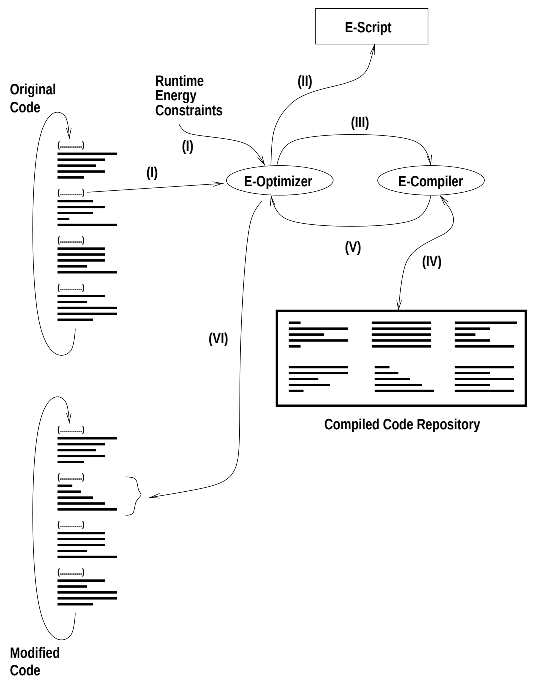

Figure 2: Updating an executable in response to a change in energy constraints. The numbers attached to arrows indicate the order of events.

2.2 Execution Model

We explain our execution model using Figure 2. In this model, the application source code is augmented using *sensitivity lists*. Each sensitivity list is attached to a loop or subprogram (function) and indicates the energy components that the following subprogram or loop is sensitive to. For example, a sensitivity list may indicate that whenever there is a change in energy constraints a specific subprogram needs to be recompiled for optimizing off-chip memory energy. In general, a sensitivity list can contain constraints that involve energy consumptions of other system components as well. Since a sensitivity list can be attached to a loop nest and function, our approach can work with both loop-based applications and function-intensive codes.

In the original code shown in Figure 2, there are four separate loop nests enclosed by a timing loop that iterates a certain number of times or till a termination condition is satisfied. Suppose that when a change in the energy constraint occurs, the execution thread reaches the second nest. Assuming that the sensitivity list of this nest indicates that the nest needs to be recompiled when such a change occurs, the execution thread temporarily stops and E-Optimizer is invoked. E-Optimizer is a decision-making module which checks E-Script to determine the compilation strategy to choose, given the new energy constraint. E-Script is a list that contains for each nest (or subprogram) the compilation strategies that should be activated based on energy constraints. Its details will be explained later in the paper. After determining the optimization strategy, E-Optimizer asks E-Compiler whether there is already a compiled module in the Compiled Code Repository that corresponds to the new energy constraint. If there is, then E-compiler supplies that module, which is subsequently inserted by E-Optimizer in the code and execution resumes. If no such a precompiled module exists in the repository, E-Compiler generates such a module and forwards it to E-Optimizer which then proceeds as explained above.

Let us now discuss a typical execution scenario where such a dynamic compilation framework can be useful. Suppose that a subprogram has been precompiled for the best performance (i.e., for minimum execution cycles). We assume that this optimized code uses a large number of memory banks during execution for data manipulation. Suppose that after some execution, the battery power is reduced significantly, which means that it is not possible to maintain power to the entire memory space used and complete application execution before battery is dead. That is, active memory space should be reduced by shutting off a number of infrequently used memory banks. In other words, the subprogram code should be modified to make it work with a smaller data space. Assuming that this variation in battery power triggers a signal which is sensed when we are checking the sensitivity list just before entering our subprogram (that is, within the routine that implements the sensitivity list check), the execution stops and E-Optimizer is invoked. Using E-Compiler, it obtains a new compiled code that has the same semantics as the previous one but can work with a smaller data memory space. This new code also shuts off the memory banks that will not be needed during execution. It should be noted that it is not sufficient to just turn off some memory banks as this can cause data loss. Instead, first, we need to apply data space minimizing optimizations, and then turn off the unused banks. E-Optimizer inserts (links) this new subprogram code into the application code and execution resumes. This version is executed until a new variation in energy constraint occurs. For example, if battery is recharged, E-Optimizer is invoked again and the previous version of the subprogram is restored. We believe that in a dynamic environment where energy constraints change during execution, such dynamic compilations and executable modifications might be extremely useful.

It should be noted that, in our implementation, the variation in energy constraint is checked only when a sensitivity list check is being done (i.e., when a sensitivity list is reached during execution). This makes our implementation easy at the expense of a longer reaction time. This dynamic recompilation/linking framework has been implemented using Dyninst. In this implementation, E-Optimizer is a separate supervisory program that controls the code modifications to be performed on the target application. In a sense, E-Optimizer corresponds to the mutator process in Dyninst and contains calls into the DyninstAPI library, whereas the application program being modified is analogous to the the mutatee process. E-Optimizer contains code to invoke our runtime compiler, E-Compiler, and the utility routines to manipulate the application process.

2.3 Tasks

The only modification to the application code is the addition of sensitivity lists to the appropriate parts of the code. Our current implementation allows sensitivity lists to be added at the beginning of subprograms and loops. For example, the sensitivity list in the following code fragment indicates that the loop nest that follows it is sensitive to off-chip memory energy (E-mem); that is, whenever there is a change in energy constraint, the loop nest should be recompiled for optimizing off-chip memory energy.

```
list (E-mem);
for (N=1;N<=4;N++)
  {
   DW[J][I][N] = 0;
   for (J=2;J<=JL;J++)
    for (I=2;I<=IL;I++)
     DW[J][I][N] += DW[J][I][N]-DU[I][J];
  }
```

In this code fragment, E-mem is termed as a specifier. In general, a sensitivity list may involve several specifiers. Our current implementation allows the following specifiers: E-mem (off-chip memory energy), E-dcache (data cache energy), E-icache (instruction cache energy), E-core (processor energy), E-interconnect (interconnect energy). A sensitivity list such as list(E-mem,E-core), for example, indicates that whenever there occurs a change in energy constraint, the subprogram (or loop) that follows this sensitivity list in the program text should be recompiled for off-chip memory energy and core energy. In our current approach, the sensitivity list has been implemented as a call to a runtime library.

In order to use our dynamic compilation environment, the user needs to perform the following tasks:

• *Identifying energy-critical regions:* The first step that needs to be performed is to identify the energy-critical regions within the application code. A region can be a subprogram or a loop. Once a region has been identified, it should be augmented by adding a sensitivity list with suitable specifiers. While identifying energy-critical regions and selecting suitable specifiers is a very important problem, it is beyond the scope of this paper. In this paper, we are mainly interested in executing an application in a dynamic recompilation/linking environment taking into account changing energy constraints.

• *Creating different versions for each critical region:* Once the energy-critical sections have been determined, the user needs to create different versions of each energy-critical region. Each version corresponds to a code which is suitable for a specific energy constraint. As an example, suppose that, for the code fragment above, the user supplies two versions: one corresponding to E-mem $<$ 500mJ and the other corresponding to E-mem $\geq$ 500mJ. What this means is that when a change in energy constraints occur if the remaining energy budget that can be allocated to off-chip main memory is less than 500mJ, the first version will be used; otherwise, the second version will be used. If the user wants, she can precompile these versions and place the compiled codes into the Compiled Code Repository. Alternately, she can just indicate in E-Script the name of the source file (all files referred in this paper are assumed to be stored in memory) and the location where the source file can be found. In the first case, using a version involves just dynamic linking, whereas in the second case, it involves both dynamic recompilation and linking.

• *Preparing an E-Script file:* The association between source codes/ compiled codes and the constraints such as E-mem $\geq$ 500mJ is done through E-Script. Each entry in the E-Script file is a tuple of the following form: [Constraints, File Name]. In such an entry, Constraint denotes the energy constraints that are written in terms of specifiers and File Name is the pointer to the source file or to the linkable (pre-compiled) code.

• *Initiating the mutator execution:* The mutator process should identify the application process that has to be modified. If the process is already in execution, the mutator attaches the mutatee process using the executable name and the process id. If, on the other hand, the application process has not yet been started, the mutator specifies the pathname and the argument list of the mutatee. The format of the command that activates mutator is:

`mutator` [-attach] <E-Script File> <Compiled Code Repository> <Application File>.

Here, the attach flag enables the mutator to supervise a process already in execution.

2.4 Version Generation

An important problem in our framework is generating different versions of an energy-critical region given an E-Script file. This is an environment-specific problem; in this study, we discuss our current version generation strategy. Our current version generation strategy is based on a compiler-assisted energy estimation model and targets a system with a banked main memory architecture. In this model, the compiler takes a loop or a subprogram, creates different versions, estimates energy consumption of each version, and associates these estimates with the runtime energy constraints using the E-Script entries.

Our main objective in using a compiler for creating multiple versions is to obtain several optimized versions of the same code quickly. We address the optimized code generation problem for two types of embedded codes: array-dominated applications and applications with more irregular data access patterns (e.g., so called integer codes). All our optimizations have been implemented using the SUIF compiler infrastructure from Stanford University [14].

In order to create different optimized versions of array-dominated loops, our compiler uses different combinations of three widely-used compiler optimizations: loop tiling, loop unrolling, and linear loop transformations [15]. We selected these optimizations as they have been shown to be effective in optimizing execution time as well as energy consumption [2]. In tiling, arrays that are too big to fit in the cache are (logically) broken up into smaller pieces (to fit in the cache) and the loop nest is restructured accordingly. Loop unrolling reduces the iteration count of a given loop by duplicating loop body (with appropriate loop iterator increments). By doing so, it aims at reducing the number of memory accesses and promoting register reuse. Linear loop transformations (e.g., loop interchange) attempt to improve cache performance, instruction scheduling, and iteration-level parallelism by modifying the traversal order of the iteration space of the loop nest.

Since innermost loop is the most critical loop in a given nest as far as energy consumption and performance of the nest are concerned, our version generator enumerates all possible innermost loops. The only loops that are not considered are the ones that violate one or more data dependences when they are placed into innermost position. All remaining loops are tried as the innermost loop. Linear loop transformations are used when it is possible to bring a loop into innermost position without violating data dependences. It should be noted that the order of outer loops is less important compared to that of the innermost loop. Consequently, in our implementation, while trying a loop in the innermost position, the order of the outer loops is kept as close to their original order as possible. For each such version, we next apply tiling to all the loops in the code (except the outermost one) that carry some form of data reuse (i.e., spatial reuse or temporal reuse). In selecting the tile size, our approach uses the strategy proposed by Coleman and McKinley [3]. Then, we focus on the outermost loop and unroll it. In this study, we use a limited set of unrolling factors (2, 4, 6, 8). Most of the codes encountered in practice work best with one of these unrolling factors. Note that this optimization strategy is a reasonable one. This is because tiling in general generates best results when it is applied to inner loops with data reuse. Similarly, loop unrolling results in the best energy and performance behavior when it is applied to the outermost loop. We also add the original loop to our version repertoire. Consequently, for a nest that contains L loops, this approach creates a maximum of $4L+1$ different versions.

For integer codes, we use different optimization strategies (for a loop or subprogram) which are similar to the optimization levels used by several commercial compilers. Our optimization levels are:

• Level-I includes local optimizations such as common subexpression elimination and strength reduction (e.g., converting multiplication to additions).

• Level-II includes optimizations that use data-flow analysis such as code motion, strength reduction, split-lifetime analysis, and instruction scheduling.

• Level-III includes all optimizations in the previous two levels as well as loop unrolling, code replication to eliminate branches, and padding certain power-of-two array sizes for more efficient use of cache.

• Level-IV includes all optimizations in Level-III plus a set of global optimizations such as inline expansion of small procedures.

• Level-V is Level-IV plus loop pipelining.

Next, for each different version, we estimate the data address regions accessed and turn off memory banks that are not accessed by that version. The objective here is to save memory energy by shutting off unused memory banks. The strategy that we use for shutting of unused memory banks is discussed in [4] and its details are beyond the scope of this paper. During this phase of version generation, the source code is also modified to insert memory bank turn off commands at the beginning of the code and bank reactivation commands at the end of the code.

After generating different versions, we estimate energy consumption of each version using the compiler. Our compiler-directed energy estimation model uses three types of parameters: technology parameters (specific constant values which depend on the process technology used), configuration parameters (e.g., memory capacity, cache topology), and program-specific parameters. The program-specific parameters are obtained by analyzing the code. The compiler estimates the number of cache misses (using the method discussed in [11]) and the number of accesses to each hardware component. Using these three parameters, the compiler then estimates the energy consumption of the code version being considered for off-chip memory, caches, interconnect, and processor core. The accuracy of this compiler-based energy estimation strategy has been shown to be within 6% of a cycle-accurate energy simulator for a single-issue, five-stage pipelined architecture. To compute (estimate) the datapath energy consumption, this compiler-based strategy estimates the number and types of assembly instructions that would be generated. The energy consumed in the caches is largely independent of the actual data accessed from the caches, and the prior work has shown that the number of cache accesses is sufficient to model energy accurately [7]. The cache hit/miss information along with the cache configuration that determines the length of the bitlines and wordlines, and the size of decoders and sense amplifiers is used to evaluate the energy consumption. The compiler also estimates the number of bus transactions and the number of accesses to the banked memory, and uses this information to estimate bus and main memory energies. The details of this compiler-based energy estimation strategy can be found elsewhere [9]. It should be noted that while it is also possible to use a simulator to obtain energy behavior of each version, that would be very time-consuming. It should also be emphasized that without estimating the energy consumption of each version, it is almost impossible to rank different versions of the code from the energy perspective. This is because different optimizations can impact energy consumption of different hardware components in complex ways [9].

After computing energy values for each version, we drop some versions from further consideration if their energy behavior is very similar to some other version. The remaining versions have different energy (and performance) behaviors and can be used to fill the entries in the E-Script file. For example, suppose that after dropping some versions, we ended up with three different versions for a given nest: $v1$, $v2$, and $v3$. Assume that we focus on off-chip memory energy consumption and the memory energy consumptions of these three versions are (estimated to be) 200mJ, 400mJ, and 800mJ, respectively. In this case, we can insert four entries to the E-Script file: $[$E-mem $\leq$ 200, $v1$ $]$, $[$ 200 $<$ E-mem $\leq$ 400, $v2$ $]$, $[$ 400 $<$ E-mem $\leq$ 800, $v3$ $]$, and $[$E-mem $>$ 800, Default $]$, where Default corresponds to the default version when no other match occurs in E-Script.

It should be stressed that while our current version generation strategy is targeted towards a system with banked memory architecture, dynamic compilation for energy can target any embedded architecture. In general, a version generation strategy can generate different versions of a given code with different performance/energy tradeoffs and based on the runtime constraints can select the best version.

Benchmark	Source/Type	Input File/Size
btrix	Spec	721 KB
g721encode	MediaBench	clinton.pcm
epic	MediaBench	test_image.pgm
full_search	Motion Estimation	310 KB
hier	Motion Estimation	310 KB
tomcatv	Spec	836 KB
vpenta	Spec	770 KB

Figure 3: Benchmark characteristics.

2.5 Implementation of Sensitivity Lists

As mentioned earlier, sensitivity list is implemented as a library routine. This library routine performs the following functionality. It first obtains the new energy constraint and, then using a predetermined energy budget distribution profile, it computes the estimated new energy constraint for each specifier in the list. It then makes a call to E-Optimizer passing this information. Our current implementation is flexible in the sense that it can be made to work with different energy budget distribution profiles. An energy budget distribution profile specifies how the new energy budget should be divided among different hardware components such as main memory, caches, processor core, and interconnect. For example, an energy budget distribution profile can indicate that the cache memories in the system should consume about 20% of the remaining battery energy. Based on this information, when a change occurs in energy constraints, the compiler calculates a target energy consumption (in absolute terms) for the caches and optimizes the code accordingly.

3. Experiments and Results

To evaluate our dynamic recompilation/linking environment, we conducted experiments using seven benchmark programs from different domains. Figure 3 gives important characteristics of these benchmark codes. In this paper, we focused only on optimizing the overall energy consumption for each code segment targeted. That is, the sensitivity list that we use contains specifiers of all hardware components considered in this work (i.e., processor core, main memory, caches, and interconnect). So, when we mention ‘energy consumption’ in this section, we mean the sum of the energies consumed in these components.

The energy consumptions reported in this section have been obtained (estimated) using the approach explained earlier. The energy spent during dynamic recompilation and linking have also been estimated by the compiler by feeding it the code that performs linking and compilation as well as the source program. In other words, our approach takes into account the overhead energy as well. The energy values are computed for a simple embedded architecture whose important parameters are listed in Figure 4. We assume that the leakage energy per cycle of the entire cache is equal to the dynamic energy consumed per access. This assumption tries to capture the anticipated importance of leakage energy in the future. We also assumed that the main memory is an SRAM and divided into 16 banks. Each bank can be individually leakage-controlled when it is not in active use. As mentioned earlier, the energy-efficient versions of the applications used in this study typically shut off some portion of the main memory. In optimizing leakage energy for the SRAM memory, we modified the voltage down converter circuit already present in current memory chip designs to provide a gated supply voltage to the memory bank. When a bank in the inactive mode is accessed to allocate a new data, it incurs a penalty of 350 cycles to service the request (a reasonable value for power supply gating). Control energy is the amount of energy required to come back from the leakage control mode. For consistency, we also assumed the same leakage energy/dynamic energy ratio for components other than main memory as well.

Simulation Parameter	Value
Processor Core	
Issue Width	1 instruction/cycle
Pipeline Width	5 stages
Functional Units	1 integer ALUs
	1 integer multiplier/divider
	1 FP ALUs
	1 FP multiplier/divider
Cycle Time	20ns
Cache & Memory Hierarchy	
Instruction Cache	16KB, direct-mapped
	32 byte blocks, 1 cycle latency
Data Cache	16KB, 2-way
	32 byte blocks, 1 cycle latency
Memory	1MB, 100 cycle latency
	16 banks (each is 64KB)
Energy Parameters	
Dynamic Energy per Cache Access (0.1μ)	0.80nJ
Leakage Energy per 32 Bytes	0.66pJ
Control Energy	0.02nJ

Figure 4: Simulated configuration.

Figure 5 presents the energy consumption of our benchmarks under three different execution scenarios. In the first scenario (the leftmost bar for each application), no dynamic recompilation or linking is performed. The application compiled and linked statically and executed. In the second scenario (the middle bar), two precompiled versions have been prepared for each benchmark: one corresponding to the default, high energy-consuming code (that works with large data memory space) and one corresponding to the energy-efficient version. Under this scenario, the execution starts with the default version. When a change in energy constraint is detected (indicating that battery energy is going down), we dynamically link the energy-efficient version. In the third scenario (the rightmost bar for each application), we have the same two versions as in the second scenario; but, this time, only the default version has been precompiled. When the energy constraint changes, we first dynamically recompile the energy-efficient version and then link it. As mentioned earlier, it is not possible to just shut off some memory banks for saving energy. We also need to restructure the program such that its data memory requirements are reduced. In Figure 5, Execution, Linking, and Recompilation portions correspond, respectively, to the energies spent during execution, dynamic linking, and dynamic recompilation (including the E-Script check). In obtaining these results, we applied dynamic compilation to the entire code in array-dominated applications and to the most frequently used functions in other codes. More specifically, in epic, we applied dynamic recompilation/linking to only internal_filter() and quantize_image(), which together constitute 88.9% of the overall execution time. In g721encode, on the other hand, we applied our strategy to three time-consuming functions: fmult(), update(), and predictor_zero().

We observe from Figure 5 that dynamic recompilation/linking has different impacts on energy behavior of different applications. In g721 encode, epic, tomcatv, and vpenta, it brings large energy benefits even if we recompile and link during execution. In hier, on the other hand, it does not perform well, primarily because the energy-efficient version does not reduce energy consumption of the default version significantly, and since dynamic recompilation and linking themselves consume energy, we see an increase in overall energy consumption. In full_search, we witness an interesting behavior. Our approach brings an energy benefit provided that we perform only dynamic linking at runtime. If we try to do both dynamic recompilation and linking, the overall energy consumption becomes larger than the original version (i.e., the one without recompilation/linking). A similar behavior is observed in btrix. In this code, the second scenario generates the

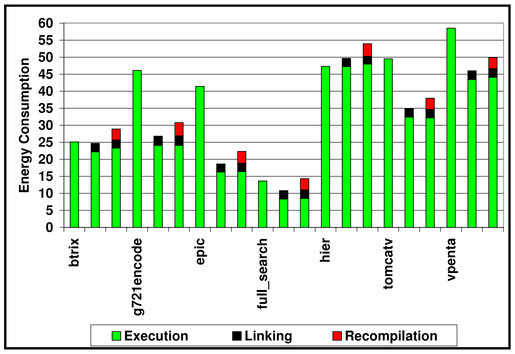

Figure 5: Energy consumptions (in millijoules) of our benchmarks under different execution scenarios.

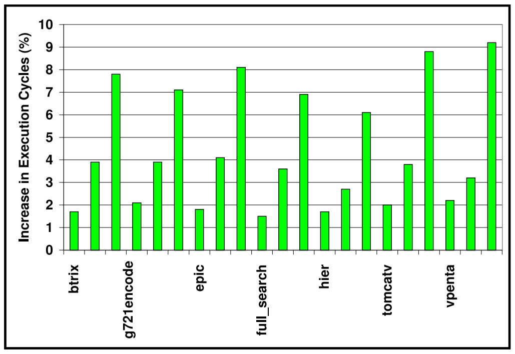

Figure 6: Increase in execution cycles (%) of our benchmarks under different execution scenarios.

same overall energy consumption as the first scenario. However, the third scenario increases the energy consumption due to extra energy expended in dynamic recompilation. Overall, these results indicate that, in some codes, large energy savings are possible (even taking into account the extra energy cost of dynamic recompilation and linking) when our approach is employed. In some other codes, on the other hand, it may not be beneficial to adopt our optimization strategy. In these cases, it might be a good idea either to energy-optimize code more aggressively, or to try to reduce the energy consumed during dynamic recompilation/linking. On average (across all benchmarks), the Linking version and the Recompilation version (in Figure 5) bring, respectively, 24.9% and 15.5% energy savings over the Execution version.

Figure 6 shows the percentage increase in execution cycles. The first two bars for each benchmark correspond to the cases dynamic linking alone and dynamic recompilation and linking used together. The increases are with respect to the default scenario where no dynamic recompilation or linking is done. We see that the average increase in execution cycles is 1.9% and 3.6% with the linking only and recompilation plus linking versions, respectively. Based on these results, we can conclude that the execution time cost of dynamic recompilation/linking is affordable. However, running a less performance-efficient version also increases the execution cycles. The third bar in Figure 6 shows the increase in execution cycles when this factor is also included (i.e., in addition to the extra cost of dynamic recompilation/linking). We observe that the overall increase in execution cycles is always less than 10% (averaging on 7.7%). We also performed experiments with varying system parameters and found that our strategy is very robust. The detailed results are omitted due to lack of space.

4. Conclusions

The results obtained so far from using our dynamic recompilation/linking strategy are encouraging. Our preliminary experiments indicate that large energy gains are possible at the expense of a relatively small increase in execution time. We strongly believe that dynamic recompilation/linking is the next step in addressing the growing energy problem in dynamic environments where energy constraints are frequently changing.

5. REFERENCES

[1] B. R. Buck and J. K. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 1994.

[2] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. DeMan. Global communication and memory optimizing transformations for low power signal processing systems. In *Proc. the IEEE Workshop on VLSI Signal Processing*, pages 178-187, 1994.

[3] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proc. the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

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

[5] K. Ebcioglu and E. R. Altman. DAISY: dynamic compilation for 100% architectural compatibility. In *Proc. the International Symposium on Computer Architecture*, 1997.

[6] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. the 23rd ACM Conference on Programming Language Design and Implementation*, 1996.

[7] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers, and bit-line segmentation. In *1999 Proc. the International Symposium Low Power Electronics and Design*, 1999, pages 70–75.

[8] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. Conference on Programming Language Design and Implementation*, May 1999.

[9] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A Sivasubramaniam. EAC: a compiler framework for high-level energy estimation and optimization. In *Proc. the 5th Design Automation and Test in Europe Conference*, Paris, France, 4–8 March, 2002.

[10] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[11] K. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[12] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), 1995, pp. 37–46.

[13] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proc. the SIGPLAN Conference on Programming Language Design and Implementation*, May 1994, Orlando, FL, pp. 196–205.

[14] R. Wilson et al. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

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