Using Lightweight Procedures to Improve Instruction Cache Performance

Krishna Kunchithapadam
University of Wisconsin-Madison
1210 West Dayton Street
Madison, WI 53706
krisna@cs.wisc.edu

James R. Larus
Microsoft Research
One Microsoft Way
Redmond, WA 98052
larus@microsoft.com

Abstract

Instruction cache performance is widely recognized as a critical component of the overall performance of a program; especially so in the case of large applications like database servers. In this report, we present a technique for (1) identifying repeated blocks of instructions in a program executable, and (2) converting these repeated code blocks into lightweight procedures (i.e. *LWprocs*).

The use of LWprocs reduces the static code size of a program, and can potentially reduce the working set size of the process, at the cost of increasing its dynamic instruction count. However, the tradeoff seems to be in favor of the reduction in working set size for most programs. Even with a simple model of program structure and a straightforward technique for generating LWprocs, we find performance improvements between 3% to 9% for programs in the *SPECINT95* suite. However, the technique sometimes leads to slowdowns (between 5% and 27%) for some programs, suggesting that lightweight procedures should be used with care.

1 Introduction

It is widely acknowledged that the instruction cache performance of a program is an important component of its overall performance. Recent studies [RGAB98, LBE+98] of workloads on large database servers find that I-cache misses have a significantly greater impact on overall program performance than comparable rates of D-cache misses. Indeed, modern out-of-order execution engines are able to tolerate the latency associated with D-cache misses far better than the pipeline stalls associated with I-cache misses. Ranganathan et al. find up to a 30% degradation in online transaction processing performance due to I-cache misses. Lo et al. report that CPI degradation due to I-cache misses and large memory footprints in database workloads may be as high as 75%.

Many techniques have been developed in the past to improve the spatial and temporal locality of execution with respect to the I-cache. Locality-oriented techniques have focused primarily on *reordering* the instructions, basic-blocks, and procedures of a program to make better use of the memory hierarchy [PH90, HKC97, CL96].

In this work, we focus on a different method for improving the I-cache performance of programs by attempting to reduce the size of their working set. We do this through a code compression technique that (1) locates repeated blocks of instructions in an executable, and (2) packs these code blocks into *lightweight procedures* or *LWprocs*. [1]

Code compression techniques similar to ours have been used in the past in embedded systems where reducing the size of a programs working set is not quite as important as merely having the code fit in memory [LDKT95, Lia96].

A lightweight procedure is a collection of instructions that may be called like an ordinary procedure from anywhere in the program. By compacting long, repeated code blocks into short procedure call sequences, we have the potential to reduce the *static size* of a program's code segment, and to also reduce the size of its *instruction working set* during execution. But this optimization comes at the cost of increasing the number of dynamic instructions the program executes (the extra instructions for `call`ing a lightweight procedure and `return`ing from it), and possible pipeline bubbles caused by the transfer of control.

In this respect, the use of LWprocs is the obverse of the technique of *procedure inlining.* Whereas procedure inlining expands a single copy of a block of instructions (the procedure) into each of its call sites, our LWproc technique identifies common blocks of instructions and collects them into a single lightweight procedure. Past work has demonstrated that procedure inlining can degrade the performance of a program by causing code size blowup, i.e. by greatly increasing the working set size of the program [DH92, DC93]. The use of lightweight procedures combines the reduction of a program's working set size with low procedure call overhead (a call to an LWproc is much cheaper than a regular procedure call).

Finally, the use of LWprocs is *orthogonal* to other locality-based I-cache optimizations, and it should be possible to combine basic-block and procedure reordering/placement with code compression techniques into a single framework.

The rest of this report is organized as follows. In section 2, we motivate the benefits of using lightweight procedures using a simple example benchmark. In section 3, we discuss the details of identifying repeated code blocks in an executable, and the use of binary rewriting to generate lightweight procedures. In section 4, we present the results of our performance study on benchmarks in the SPECINT95 suite. In section 5, we discuss two extensions to the a model for identifying and generating lightweight procedures. In section 6, we compare our work to other techniques for improving the I-cache performance of programs. Finally, we close with a summary of our work.

[1] We use the term *LWprocs* rather than *LWP* to avoid confusion with the well established meaning of *lightweight process* for the latter term.

2 Motivation

In order to motivate the usefulness of lightweight procedures, we will first discuss a synthetic benchmark. This benchmark has been constructed to demonstrate an extreme case of program structure—namely, one in which the use of procedures is advantageous (to the point of improving the execution time of the program by a factor of 3). Thus the synthetic benchmark gives us an upper limit (of sorts) on the kind of performance improvements that we can expect from the identification of common code and the use of lightweight procedures in more realistic programs.

We present the optimizations in terms of source code changes for clarity in illustration. In the section 3, we shall discuss the use of binary rewriting to identify repeated code blocks and to generate lightweight procedures in application executables.

Consider the C program in figure 1.

```
static void p0() {
    goto L0;
L0: NOPS; goto L9;
L1: NOPS; goto L8;
L2: NOPS; goto L7;
L3: NOPS; goto L6;
L4: NOPS; goto L5;
L5: NOPS; goto LL;
L6: NOPS; goto L4;
L7: NOPS; goto L3;
L8: NOPS; goto L2;
L9: NOPS; goto L1;
LL: ;
}

int main() {
    int i; for (i = 0; i < 10000; ++i) {
        p0();
    }
}
```

Figure 1: Program `proc0`

In the above code `NOPS` represents a large number of `nop` instructions, *large enough* that although each block of NOPS fits in the I-cache, the entire procedure `p0()` is larger than the size of the I-cache. [2] Therefore, any execution of the `proc0` will cause a large number of I-cache misses.

[2] The jumps have been added to `p0()` to break up the procedure body into smaller basic-blocks.

The same program can be rewritten as shown in figure 2.

```
static void lwproc() {
    NOPS;
}

static void p1() {
    goto L0;
L0: lwproc(); goto L9;
L1: lwproc(); goto L8;
L2: lwproc(); goto L7;
L3: lwproc(); goto L6;
L4: lwproc(); goto L5;
L5: lwproc(); goto LL;
L6: lwproc(); goto L4;
L7: lwproc(); goto L3;
L8: lwproc(); goto L2;
L9: lwproc(); goto L1;
LL: ;
}

int main() {
    int i; for (i = 0; i < 10000; ++i) {
        p1();
    }
}
```

Figure 2: Program `proc1`

Program `proc1` is functionally equivalent to program `proc0`; however, the combined size of both `lwproc()` and `p1()` is now much smaller than the size of the I-cache. Each block of NOPS in `p0()` has been replaced with a call to the procedure `lwproc()` (which is a *lightweight* procedure in our terminology), and there is only one copy of `lwproc()` in the entire program.

While program `proc1` will execute many more dynamic instructions than program `proc0` (to be specific, `proc1` will execute at least 2 extra dynamic instructions—one call and one return—per execution of a block of NOPS, compared to `proc0`), it is also the case that `proc1` will execute completely inside the I-cache of the machine, and can therefore be expected to have a much smaller execution time than `proc0`. What is more, `proc1` has a much smaller static text segment size than `proc0` since all of the repeated NOPS code blocks have been collected into a single procedure `lwproc()`.

The table in figure 3 shows the static size of the text segments, execution times, and the dynamic instruction counts and I-cache miss rates (as measured by the Shade [CK94] analyzer) for programs `proc0` and `proc1`. [3]

Program	Text size (in bytes)	User time (in seconds)	#instructions	I-cache miss rate
proc0	171779	2.52	410248065	12.51%
proc1	24419	0.87	410948065	0.21%

Figure 3: Code size, execution time, dynamic instruction count, and I-cache miss rates for `proc0` and `proc1`.

As can be seen, in spite of the increased number of dynamic instructions executed by program `proc1` (and in fact, the increase in dynamic instruction count is a mere 0.17%), it has a substantially smaller execution time (smaller by a factor of about 3) and I-cache miss rate.

This leads us to suspect that the use of lightweight procedures will be advantageous in more realistic programs.

There is another difference between `proc0` and `proc1` that merits some discussion here. The difference is caused by the nature of the additional instructions executed by `proc1`. As mentioned before, and as can be seen from the use of a procedure call, all of the additional instructions in `proc1` are branches (the `call` instruction to a lightweight procedure and its `return` counterpart). Control-transfer instructions have the potential to cause performance degradations in deeply pipelined CPUs, and it is all the more surprising that `proc1`, despite its higher dynamic instruction count *and* higher rate of executing branch instructions, has a better performance than `proc0`.

However, the supposed negative performance impact of the additional `call` and `return` instructions in `proc1` is, we believe, an illusion and one whose impact is almost certain to be less significant with each succeeding generation of processors. The reason for this view is that the additional control-transfer instructions in `proc1` are of a very special kind—they are *non-conditional* branch instructions with, in the case of the `call` instruction, a known target. Such instructions are especially easy for a processor to handle via *branch-prediction* and *instruction prefetching* techniques and should cause minimal disruptions to the execution pipeline. Similarly, a `return` instruction is equally easy to handle and many micro-architectures support (or have proposals to add) a stack that mirrors the return addresses of the procedure call hierarchy. Moreover, these control-transfer instructions always occur in the context of computational instructions and CPUs with multiple execution units and multi-way issue logic can execute the branch instructions in parallel with the regular computation instructions.

Our evidence for the above conjectures is only indirect (since we do not have access to a detailed micro-architectural simulator for the SPARC processor).

[3] The executables were dynamically linked to highlight the differences in the text sizes of the code without the large runtime libraries. The programs were run on a 250 Mhz UltraSPARC-II, 16kB I-cache, 32kB D-cache, running Solaris-2.6

However, in section 5, we shall present examples of micro-benchmarks where there seems good reason to believe that the additional control-transfer instructions are executed almost for free, with no negative impact on the performance of the program.

Having provided the motivation for the magnitude of benefits that can be obtained from using lightweight procedures, we shall now discuss the detailed technique for generating lightweight procedures in programs without any reference or access to source code, i.e. by using binary rewriting techniques, and the performance of these modified programs.

3 Generating Lightweight Procedures

In order to identify, generate, and use lightweight procedures, we need to address the following problems:

- Identify repeated code blocks in the program.
- Select, from these repeated code blocks, those suitable for encapsulating into a lightweight procedure (where the selection will be made based on both the size and the frequency of occurrence of the code blocks).
- Generate lightweight procedures from the selected candidates.
- Modify the program to reorder its procedures and patch the code to make calls to the lightweight procedures as appropriate.

While it is possible to do all of the above steps at the source-level, we use the technique of binary analysis and rewriting [LB92, BL92, LB94] to perform each of the above actions. Working at the level of an executable offers significant advantages over source code:

- Repeated code blocks can be identified across the entire program, rather than within a single procedure, source file, or module.
- Both the application code and libraries can be analyzed.
- Repeated code blocks that are not apparent at the level of source code (and code sequences which are specific to machine architecture) can be identified.

The Executable Editing Library (EEL) [LS95, Lar97] allows us to build custom binary rewriting applications, while handling all of the low-level system-specific details of reading an executable file, analyzing its contents to recover the control and data flow structure of the program, and writing a modified executable in the appropriate format. We use EEL-based applications in all of our work on LWprocs.

In the rest of this section we shall describe our approach to identifying and generating lightweight procedures. The technique uses an extremely simple cost model and does not take a program's loop structure into account. More complex models can be defined which have the potential to perform better on a wider variety of applications (we shall discuss some of these extensions later). Since we use EEL, which currently handles only the SPARC instruction set architecture, some of the details in this section will be specific to SPARCs. However, the general techniques are more widely applicable.

3.1 Identifying repeated code blocks

The first step, i.e. the identification of repeated code blocks, works at the granularity of basic blocks in the program. [4] Using EEL, we identify all single-entry, single-exit code blocks in the program and select from among these blocks those that satisfy the following conditions:

- The block occurs within a *non-leaf* procedure. Leaf procedures occur at the leaf nodes of a call-graph hierarchy and are optimized to not have a stack frame (they use the stack frame of the calling procedure). This optimization precludes a leaf procedure from making procedure calls—hence we cannot generate lightweight procedures from code blocks that are inside a leaf procedure, even if they are repeated elsewhere in the binary.

- The instructions in the code block do not read or write the procedure return register (%o7 on the SPARC). If a code block is converted into a lightweight procedure, all of its instructions will move to a new location in the text segment of the modified program. Hence an instruction that depends on a specific value for the PC (or uses PC-relative addressing) cannot be moved into a lightweight procedure. Likewise, instructions which modify the return address register cannot be moved into a lightweight procedure—otherwise the modified program would have different semantics than original. [5]

- Finally, if a basic block terminates in a control transfer instruction (i.e. a branch or a procedure call), this instruction (and its delay slot instruction) cannot be moved into a lightweight procedure. However, all of the preceding instructions in the basic block are potential LWproc candidates.

  The reason control-transfer instructions cannot be moved into a lightweight procedure is similar to the reason that prevents us from moving other PC-relative instructions, i.e. branches and procedure calls are addressed

[4] An obvious extension to this approach would be to attempt to search for repeated *subgraphs* of a program's context flow graph. However, our tests indicate that as we widen the scope of search beyond the granularity of basic-blocks, the opportunities for finding repeated instructions drops dramatically. The issue is discussed in fuller detail in section 5.

[5] EEL is clever enough to relocate instructions with PC-relative addressing by changing the index offsets. However, this relocation will not work for LWprocs since multiple copies of an instruction are being folded into a single copy when a lightweight procedure is generated. A single, possibly relocated, PC-relative offset cannot correctly implement the semantics of multiple instructions located at different addresses in the original program.

relative to the program counter. Since multiple copies of an instruction are merged into a single copy inside an LWproc, no single value of a PC-relative offset be correct for all calling points.

It is also possible to split a single basic-block into smaller blocks and convert the sub-blocks into lightweight procedures, although we have not examined this option in our current work.

Each basic block (excluding any control-transfer instructions) that satisfies the above conditions is considered a potential candidate for being converted into a lightweight procedure. Such conversions are safe (a modified program where each *safe* basic-block was replaced by a call to a suitable lightweight procedure will have the same semantics as the original program). However, converting every candidate code block into a separate lightweight procedure, is, obviously, a silly thing to do. It makes sense to convert only those blocks that are repeated *often enough* (in the sense defined below).

Figure 4 shows a program with multiple procedures and multiple repeated code.

```
procedure p0:   procedure p1    procedure p2
    ...             ...             ...
L0:             L3:             L7:
    instr0          instr3          instr3
    instr1          instr4          instr4
    instr2          b L5            b L9
    b L2        L4:             L8:
L1:                 instr3          instr0
    instr3          instr4          instr1
    instr4          b L6            instr2
    b L1        L5:             L9:
L2:                 instr6          instr7
    instr5          ...             ...
    ...
```

Figure 4: Original program with repeated code blocks

The set of instructions in figure 4 can be modified to use lightweight procedures as shown in figure 5.

Figure 5 demonstrates the salient features and optimization of our LWproc generating technique. We reduce the overhead of invoking a lightweight procedure to the minimum of 2 instructions (a `call` and a `retl` (the leaf-return instruction on the SPARC)) by filling the delay slots of the control-transfer instructions with instructions from the original code blocks. This optimization also provides us with our static cost model for selecting LWprocs (as discussed below). However, we need to point out that the above example is meant merely to illustrate the nature of the rewriting transformation involved in generating

```
procedure lwp0      procedure lwp1
    instr1              retl
    retl                instr4
    instr2

procedure p0:   procedure p1    procedure p2
    ...             ...             ...
L0:             L3:             L7:
    call lwp0       call lwp1       call lwp1
    instr0          instr3          instr3
    b L2            b L5            b L9
L1:             L4:             L8:
    call lwp1       call lwp1       call lwp0
    instr3          instr3          instr0
    b L1            b L6        L9:
L2:             L5:                 instr7
    instr5          instr6          ...
    ...             ...
```

Figure 5: Program modified to use LWprocs

and using lightweight procedures—the need for brevity in the example means that the code presented here will not conform to the cost model that we discuss below.

3.2 Selecting lightweight procedures

As the executable is being scanned, all the instructions of each candidate code block are stored in a hash multiset. When the entire binary has been scanned, the multiset has information about the *size* of the various candidate code blocks, and the *number of copies* of these blocks that occur in the program.

We eliminate from the multiset all code blocks which occur only once in the program—these are not suitable for converting into lightweight procedures.

From among the remaining blocks we choose those that satisfy the following equation:

$$C * N > (2 * C + N) \tag{1}$$

where C is the number of copies of a given repeated code block and N is the number of instructions in the block.

The left hand side of the above inequality represents the total size of candidate copies of a given block in the original program (C copies, each with N instructions). The right hand side of the inequality represents the total size of the lightweight procedure that can generated from these candidate blocks (N instructions; $N-1$ from the original code block and one `retl` instruction) and the replacement of each of the repeated blocks with a call to the single lightweight procedure (C copies, each with 2 instructions—a call instruction and its delay slot filled with an instruction from the original code block).

The inequality determines if it is possible to reduce the static size of the code segment of the program by moving repeated code blocks into a single lightweight procedure, and captures the kind of optimization represented in figure 5. Clearly, if there are many repeated copies (large C), or the repeated blocks have many instructions (large N), it is advantageous to generate lightweight procedures from these blocks.

The above cost model does not account for the loop structure of the program. In section 5 we discuss extensions to the above model to account for loop structure, and whether such extensions provide any additional performance benefits compared to the simpler model.

3.3 Generating LWprocs

Once all of candidate code blocks in the hash multiset have been analyzed to find suitable LWprocs (i.e. those that reduce the static code size of the program), we use EEL to scan and rewrite the binary.

EEL can package any sequence of instructions into a new procedure and introduce this procedure into an executable. In addition, EEL can make calls to these newly created procedures from any other part of the original program, and patch these call instructions as needed (depending on the exact addresses of the calling instruction and the called procedure). We simply use these features of EEL to package the candidate code blocks that satisfy our cost model into lightweight procedures. We also use EEL to replace the code blocks in the original program corresponding to a lightweight procedure with a call to the LWproc.

Before we produce a new executable, we perform one final optimization. EEL has the ability to reorder the procedures of an executable according to any specification. We use this feature in EEL to co-locate each procedure and all the LWprocs that it calls so as to maintain spatial locality in the program. However, this co-location cannot be done for all procedures and all LWprocs. For example, if procedure *P1* and *P2* make a call to LWproc *lwp0*, and if *P1* and *P2* are located far apart in the program executable, it is impossible for *lwp0* to be located spatially close to both procedures.

We therefore use profile information to rank the procedures in the program in decreasing order of cumulative execution time, and co-locate an LWproc with a procedure of higher profile rank. [6]

The pseudo code below describes the heuristic:

[6] An alternative to co-locating procedures and LWprocs would be to clone the LWprocs—we did not use this method since our goal is to reduce the static code size of the program. Furthermore, it is necessary to co-locate procedures and LWprocs only w.r.t. cache pages, not memory pages. A more sophisticated I-cache layout scheme in conjunction with the co-location heuristic would be advantageous.

```
foreach procedure p (sorted by decreasing profile counts) {
    foreach lwproc l called by p {
        if (l is not already generated) {
            generate l;
        }
    }
    generate p with calls to appropriate lwprocs;
}
```

Despite the fact that the above technique is extremely simple (repeated code blocks are identified only at the granularity of basic-blocks), and that we use a simple static cost-model (reduction in static size of the program's code segment), our experiments show that there are significant performance improvements to be gained even from many programs in the SPECINT95 suite.

4 Performance Results

In this section, we present the results of the use of lightweight procedures on a set of benchmarks (which include a couple of small C++ programs, and a subset of programs in the SPECINT95 suite).

Since we use the EEL toolkit to perform binary rewriting, it is not completely meaningful to directly compare the execution times of the original program and the rewritten one—the rewriting modifications performed by EEL introduce some overheads that can mask any performance improvements in the binaries due to LWprocs. [7] We do report the execution times of the original programs for comparison.

In addition to the overheads associated with binary rewriting, the ordering of the procedures in the program that uses LWprocs is different from the original ordering of procedures—as described in the previous section, this reordering is done so that each lightweight procedure is located spatially close to the procedures that call it. Merely reordering the procedures of a program (without using LWprocs) can lead to a change in performance, once again masking the effects of the use of LWprocs.

To isolate the effects of using LWprocs, we perform the following steps:

- Instrument each program in our test suite using the EEL-based path profiler PP [BL96].
- Collect performance information from the execution of the instrumented programs, and rank the procedures of the program in decreasing order of execution time. Such a profile is similar to that produced by `prof` or `gprof`.

[7] There are some instances where merely rewriting an executable, without any attempt at reordering procedures or basic-blocks, can result in a new executable than runs significantly faster than the original.

- Rewrite the original program using EEL, with the procedures ordered according to their profile rank (however, no other modification or optimization is done). This new executable is the *Null Rewrite* and is used as the baseline for comparisons.

- Analyze the original program to locate repeated code blocks, select and generate lightweight procedures (based on our cost model), and rewrite a new executable with the procedures ordered by their profile rank, and with calls to LWprocs where appropriate. The performance of this executable, the *LWproc Rewrite*, is compared with that of the Null Rewrite.

The reordering of the procedures in the Null Rewrite based on profile rank accounts for any random changes in the performance of the benchmarks that might result from simply changing the position of the procedures in a rewritten executable. Both the Null Rewrite and the LWproc Rewrite should, in principle, benefit or degrade in performance equally from any such random effects.

All our tests were carried out on a 250Mhz UltraSparc-II, with a 16kb I-cache (2-way associative, 32 byte lines), 16kB D-cache (direct mapped, 32 byte lines, 2 sub-blocks), and running the Solaris-2.6 operating system. User times (in seconds) are reported.

The benchmarks that we test include two small C++ applications (compiled with GNU g++ at the highest levels of optimization to force the compiler to inline methods as aggressively as possible), and all of the SPECINT95 programs except *gcc* and *li*. [8] Aggressive inlining might positively impact the LWproc technique for the C++ programs, but does not favor the use of LWprocs for the SPECINT95 C programs. In any case, the use of high levels of optimization is appropriate as a baseline for comparison since real-world programs are always compiled with optimization, and the benefit of any incremental technique like ours is in any marginal benefit that it provides over and above what the compiler already does.

All of the programs were compiled with the GNU gcc/g++ compiler (version 2.8.1) and linked as static executables so that repeated code in the standard C language and system libraries would also be included in the search for repeated code blocks and LWprocs.

The SPECINT95 programs were compiled for the *peak* configuration and timed on the *test* data sets. The path profiles and procedure ranks were computed from data collected by running the *peak* binaries on the *train* data set.

Figure 6 reports the size of the rewritten text segments in the Null Rewrite and the LWproc Rewrite, the percentage reduction in the static sizes, the number of lightweight procedures, the number of static LWproc call sites, the average size an LWproc, and the average number of instructions saved per static LWproc call (i.e. savings weighted by the number of static call sites for each LWproc) as generated by our hashing technique.

[8] The EEL toolkit encountered some problems analyzing and rewriting these two benchmarks, and we hope to fix the problem soon.

Program	Null size (bytes)	LWproc size (bytes)	Percent reduction	LWproc count (static)	Call sites (static)	Avg. size (bytes)	Avg. savings (#instr.)
anagram	186588	176932	-5.18	122	715	29.9	19.8
simulate	171784	162672	-5.30	85	596	29.1	26.8
go	402800	399384	-0.85	186	610	28.5	4.6
m88ksim	269064	265888	-1.18	135	475	31.5	5.9
compress	180912	179692	-0.67	57	229	28.2	5.4
ijpeg	360060	354324	-1.59	177	809	27.7	8.1
perl	439368	430516	-2.01	238	1278	25.6	9.3
vortex	634648	619608	-2.37	419	3035	26.0	9.0

Figure 6: Code sizes and Average LWproc size in bytes; average savings per LWproc in number of static instructions per call site.

As can be seen from figure 6, the two C++ programs (which had been compiled with aggressive inlining) show a greater percentage reduction in code size from the use of LWprocs. The C programs in the SPECINT95 suite show a more moderate percentage reduction in code size. However, there does not seem to be any significant correlation between the percentage reduction in size in the various programs and the number of lightweight procedures selected by the cost model. Although the average size of LWprocs is fairly uniform over all programs, the number of calls to the LWprocs varies and hence there is a marked difference in the average number of instructions saved for every LWproc generated.

Figure 7 reports the performance of the Null and LWproc rewrites in terms of the user times of the programs.

Program	Original user time (secs)	Null user time (secs)	LWproc user time (secs)	Percent reduction
anagram	2.56	2.40	2.28	-5.00
simulate	48.19	71.44	69.83	-2.25
go	167.7	176.13	184.97	5.02
m88ksim	726.5	391.15	355.90	-9.01
compress	222.43	240.90	229.88	-4.57
ijpeg	173.1	178.62	172.70	-3.31
perl	151.49	177.63	172.00	-3.17
vortex	462.84	407.49	520.49	27.73

Figure 7: User time in seconds; % reduction in LWproc time compared to Null Rewrite time.

Except for *go* and *vortex*, the use of LWprocs improves the performance of all the SPECINT95 benchmarks by anywhere from 3% to 9% We are not quite sure of the cause for the serious performance degradation in *go* and *vortex*, although I-cache interference may play a role. In such cases, the combination of LWprocs with better layout algorithms (e.g. the cache coloring scheme of Hashemi et al. [HKC97]) would help. The two C++ programs also show a significant performance improvement.

Seeing any performance improvement at all for SPEC programs is, to some extent, surprising since manufacturers design processors, caches, and systems to perform well on these benchmarks. These results are quite encouraging and suggest that larger programs (like database servers) which have large working set sizes are quite likely to benefit even more significantly from the use of lightweight procedures in terms of improved I-cache and overall execution performance.

Due to their sheer size and complexity, large programs (and commercial database servers, in particular) are not easy to edit with EEL (for example, EEL's relocation techniques either break down or lead to a high overhead for extremely large code segments). These programs are also often built from shared libraries, which EEL does not handle fully, as yet. We are working with compiler writes and vendors [Kun97] in adding annotations to program executables that would allow EEL (and similar tools) to analyze and rewrite these programs with greater ease.

5 Extensions to Models for Generating LWprocs

As mentioned in section 3, there are many possible extensions to the simple model for identifying and generating LWprocs.

In this section, we explore two of them, namely: (1) accounting for the loop structure of a program's control-flow graph, and (2) extending the notion of repeated code blocks to extended basic blocks.

5.1 Accounting for Program Loop Structure

In section 2 we considered two programs `proc0` and `proc1`, and showed how the use of lightweight procedures can dramatically improve the performance of `proc1` over `proc0`.

However, few programs have the special structure of `proc0` or `proc1`. Most real-world programs are built out of loops. It is possible to construct a program that executes about the same number of dynamic instructions as `proc0`, and yet executes almost as quickly as `proc1`.

Consider the C program in figure 8.

In program `loop0`, the "loop" in the main() function of `proc0` has been pushed into the body of the procedure `l0()`. Even though `l0()` is of the same size as `p0` (perhaps even slightly larger due to the overhead of loop management), `l0()` exhibits a temporal locality that allows it to execute completely within the I-cache. It is possible to "un-inline" the NOPS basic-blocks in `loop0` as we did with `proc0`.

The program in figure 9 is the result.

The table in figure 10 shows the static size of the text segments, execution times, the dynamic instruction counts and I-cache miss rates of `loop0` and

```
#define LOOP for (i = 0; i < 10000; ++i) { NOPS; }

static void l0() {
    int i; goto L0;
L0: LOOP; goto L9;
L1: LOOP; goto L8;
L2: LOOP; goto L7;
L3: LOOP; goto L6;
L4: LOOP; goto L5;
L5: LOOP; goto LL;
L6: LOOP; goto L4;
L7: LOOP; goto L3;
L8: LOOP; goto L2;
L9: LOOP; goto L1;
LL: ;
}

int main() {
    l0();
}
```

Figure 8: Program loop0

`loop1`.

What is surprising in these performance numbers is that there is almost no difference in the execution times of `loop0` and `loop1`, in spite of the fact that `loop1` executes many more dynamic instructions than `loop0` and `loop1` has the kind of program structure with temporal locality that would not, at first glance, seem to benefit from the use of LWprocs.

It seems as if the use of LWprocs is orthogonal to the temporal locality (or lack thereof) properties in the program. Indeed, if a block inside a loop is converted into an LWproc, then the new program still retains the temporal locality present in the original program.

The new program also does not suffer a performance degradation due to the extra dynamic instructions since (as discussed in section 2), these additional instructions are special kinds of branch instructions that can be easily predicted.

The possible loss of spatial locality in the program by the conversion of a block into an LWproc is also mitigated by co-locating the LWproc with the procedure that uses it.

It therefore seems that a more complex cost model that accounts for the loop structure of the program, and selects lightweight procedure candidates only from those regions of the code not embedded in loops, is unlikely to yield any additional performance improvements. The simple cost model is sufficient.

We implemented a modification to our LWproc-rewriting algorithm that ig-

```
#define LOOP for (i = 0; i < 10000; ++i) { NOPS; }

static void lwproc() {
    LOOP;
    return;
}

static void l0() {
    int i; goto L0;
L0: lwproc(); goto L9;
L1: lwproc(); goto L8;
L2: lwproc(); goto L7;
L3: lwproc(); goto L6;
L4: lwproc(); goto L5;
L5: lwproc(); goto LL;
L6: lwproc(); goto L4;
L7: lwproc(); goto L3;
L8: lwproc(); goto L2;
L9: lwproc(); goto L1;
LL: ;
}

int main() {
    l0();
}
```

Figure 9: Program loop1

nored LWproc candidates occurring inside *small* [9] program loops from consideration for conversion into lightweight procedures, even if the simple cost model would have earlier considered them. LWproc candidates occurring in larger program loops were still considered for conversion, as were candidate blocks that were not inside a loop in a procedure. We did not consider cases of program loops which themselves contain procedure calls—these procedures are not different in terms of either temporal locality or spatial locality than lightweight procedures.

In these experiments, the performance of the various SPECINT95 benchmarks did not change compared to those figures reported in section 4. A loop-aware cost model does not provide any additional performance improvement for these benchmarks, and, equally, does not degrade performance either.

The pairs of programs `proc0`/`proc1` and `loop0`/`loop1` represent the range

[9] The meaning of *small* depends on the size of the I-cache of the machine on which the rewritten program is meant to be executed. We used a size of 16kB, in line with that of the actual machine we used for our experiments.

Program	Text size (in bytes)	User time (in seconds)	#instructions	I-cache miss rate
loop0	172675	0.866	410782566	5.12%
loop1	24419	0.864	410782636	0.12%

Figure 10: Code size, execution time, dynamic instruction count, and I-cache miss rates for `loop0` and `loop1`.

of program structures in almost all real-world code. Procedure-oriented codes exhibit little spatial or temporal locality, and have the potential to benefit from the identification and un-inlining of repeat code blocks. Database servers are examples of procedure-oriented programs.

Loop-oriented codes exhibit a high degree of spatial and temporal locality, especially if the size of the loops fit inside the I-cache of the machine. Such programs are unlikely to benefit from the use of lightweight procedures. However, the use of LWprocs in even such programs is not likely to degrade performance.

In both kinds of programs, the simple cost model that was discussed in section 3 seems to capture the benefits of identifying and using lightweight procedures.

5.2 Using Extended Basic Blocks as LWproc Candidates

The algorithm for identifying LWprocs presented earlier (in section 3) works at the granularity of basic-blocks, i.e. single-entry, single-exit sequences of instructions without any internal branches.

We considered the effects of extending the scope of LWproc candidates to the granularity of extended basic blocks, i.e. sequences of instructions which are still single-entry, single-exit, but possibly with branches among the various sub-blocks of the extended block.

An extended basic-block as defined above is a subgraph of the control-flow graph that is bounded by a dominator and a post-dominator node, i.e. the root block of the subgraph dominates all of the blocks in the extended block (which means that the only way for control-flow to reach any of the blocks in the extended block is to first go through the root block), and that the tail block of the subgraph post-dominates all of the blocks in the extended block (which means that the only way for control-flow to exit the extended block is to finally go through the tail block).

The above, rather strict, restrictions are necessary for an extended block to be even considered a candidate for being transformed into a lightweight procedure. These restrictions are necessary since an LWproc will not be situated in the same set of addresses as the original block of instructions—if there had been a jump into an extended block that did not first go through the root block, such a jump would, in the case of LWproc, transfer control to a region of code with no way to preserve the original semantics of the program. Likewise, a jump out of the extended block that did not finally go through the tail block would, in the case of an LWproc, lead to control flow that does not preserve the semantics of the original program. In this respect, the dominator/post-dominator requirement for LWproc candidates is akin to specifying that it is illegal (or at least bad form) in a program to directly into or out of the body of a procedure without the usual setup and cleanup code associated with procedure call and return respectively.

The strict requirements are necessary to perform a correct transformation of an extended basic block into an LWproc. However, these very requirements also mean that the number of extended blocks that satisfy these conditions will be greatly reduced from the number that would be considered without these restrictions. Moreover, the reduction in the number of LWproc candidates also means that there is a reduced chance of the candidate extended blocks of satisfying the requirements of the cost model (even of the simple cost model given in section 3).

We implemented the above dominator/post-dominator search algorithm in our framework for identifying LWproc candidates and select from among them, those blocks that would satisfy our cost model. The table in figure 11 shows the number of LWprocs (of extended basic blocks with more than the trivial one basic block), the average size of these LWprocs, and the number of call sites to these LWprocs.

These numbers do not include LWprocs composed of single basic blocks and hence highlight the possible incremental benefit of considering extended basic blocks as LWproc candidates.

Program	LWproc count (static)	Call sites (static)	Avg. size (bytes)
anagram	7	24	8.5
simulate	8	26	8.8
go	6	17	8.7
m88ksim	3	6	20
compress	0	0	(n/a)
ijpeg	4	8	13
perl	7	36	5.1
vortex	4	54	2.2

Figure 11: Number of LWprocs, number of static call sites, and average size per LWprocs in number of static instructions per call site.

As can be seen from figure 11, the number of LWprocs generated from extended basic blocks is more than an order of magnitude smaller than the number of LWprocs generated with basic blocks. Moreover, the strict requirements on the extended blocks greatly reduces (often by more than *two* order of magnitude) the number of copies of these candidates that satisfy the cost model. The average number of instructions saved by the use of such LWprocs is also much smaller than the equivalent figure for LWprocs composed of basic blocks.

It therefore seems that even this extension to the simple LWproc model, of considering extended basic block candidates, does not provide any additional benefit in terms of reduction in the static code size of our benchmarks. We did not perform the rewriting transformations to generate new binaries or measure their execution times since the above numbers were so discouraging.

5.3 Summary of extensions

In this section, we considered the two most promising extensions to our simple scheme for identifying and generating LWprocs. Both extensions turned out to not provide any added benefit in performance. Underlying this somewhat discouraging message is the fact that the technique of using LWprocs is insensitive to the loop structure of the program, and can hence be employed in its simple form over a wide range of programs.

6 Related Work

The earliest memory-hierarchy optimization techniques were used on systems with small amounts of physical memory. In the past, programmers have examined source code to identify common blocks of code and moved them into procedures (in effect performing the identification and generation of LWprocs by hand) [Gra97]. Brenda Baker [Bak95] presents algorithms based on fast string-matching that can discover exact and parameterized duplication in source code—thereby enabling the identification of procedures even when variables, for example, have different names.

Modern locality-based techniques owe their origin to Pettis and Hansen [PH90] who identify the two main kinds of optimizations based on the placement of (1) procedures and (2) basic-blocks. Hashemi, Kaeli, and Calder [HKC97] extend this work with a coloring algorithm to account for varying cache sizes and associativities. Cohn and Lowney [CL96] use profile information to classify basic-blocks as *hot* or *cold* and use this information to guide locality-based reordering.

Our work does not depend on a locality-based optimization technique nor does our cost-model depend on size or the organization of the I-cache. However, all of the above mentioned technique are orthogonal to our own and can be combined into a single framework for I-cache optimization.

Fraser, Myers, and Wendt [FMW84] discuss a technique of analyzing assembly instructions to discover potential *procedure abstractions* (which they regard as the obverse of procedure inlining) in order to facilitate code *compression*, but without specific reference to the dynamic memory footprint or the I-cache performance of the resulting code. Our work focuses on the analysis and discovery of lightweight procedures in the context of I-cache performance.

Stan Liao [LDKT95, Lia96] discusses techniques for the generation of lightweight procedures in the context of embedded systems with limited memory. This work examines code at the granularity of extended basic-blocks and uses a dictionary-based compression algorithm to create LWprocs (referred to as *mini-subroutines* in the work). Liao also works from the control-flow graph of the program *before* code generation and uses code compression to guide the code generator.

Our technique is very similar to this work although we do not examine code at a granularity beyond the basic-block level. The advantage of doing so is that our technique has minimal overhead and we do not need to generate any patchup code or extra basic blocks.

7 Conclusion

In this report, we present a technique for improving the instruction cache performance of programs that is based on the reduction of working set sizes. Our technique identifies repeated blocks of instructions in an executable, and uses binary rewriting to convert these repeated code blocks into lightweight procedures (LWprocs) using a static cost model. Thus our technique differs from the commonly used locality-based methods of optimizing I-cache performance.

Experiments with the SPECINT95 suite of programs indicate that our technique can improve the performance of these benchmarks by 3% to 9%. We expect to see even more significant performance improvements on large applications like database servers. However, the use of lightweight procedures is not always a win—some programs degrade in performance, and the technique should be used with care.

Using LWprocs is also orthogonal to locality-based optimizations and may be combined with them—with the addition of a cost-model based on dynamic profile information and program loop-structure, we believe that the use of lightweight procedures will become an important component in the toolkit of I-cache optimization techniques.

References

[Bak95] Brenda S. Baker. Parameterized pattern matching by Boyer-Moore-type algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, San Francisco, California, 22–24 January 1995.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Principles of Programming Languages*, January 1992.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIG-*

METRICS Conference on the Measurement and Modeling of Computer Systems, pages 128–137, May 1994.

[CL96] Robert Cohn and P. Geoffrey Lowney. Hot cold optimization of large Windows/NT applications. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 80–89, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[DC93] J. Dean and C. Chambers. Training compilers to make better inlining decisions. Technical Report TR 93-05-05, University of Washington, 1993.

[DH92] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Trans. on Softw. Eng.*, 18(2):89, February 1992.

[FMW84] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*, pages 117–121, Montreal, Canada, 1984.

[Gra97] Jim Gray. In a personal conversation with james larus. 1997.

[HKC97] Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 171–182, New York, June15–18 1997. ACM Press.

[Kun97] Krishna Kunchithapadam. Sun microsystems internal report. 1997.

[Lar97] James R. Larus. *EEL Guts: Using the EEL Executable Editing Library*, July 1997.

[LB92] James R. Larus and Thomas J. Ball. Rewriting executable files to measure program behavior. Technical Report 1083, Computer Sciences Department, University of Wisconsin-Madison, March 1992.

[LB94] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 24(2):197–218, February 1994.

[LBE+98] J. Lo, L. Barroso, S. Eggers, K. Gharachorloo, H. Levy, and S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 39–51, New York, June 27–July 1 1998. ACM Press.

[LDKT95] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer Aided Design*, pages 393–401, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[Lia96] Stan Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Los Alamitos, Ca., USA, January 1996.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, La Jolla, California, 18–21 June 1995.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 16–27, White Plains, NY, USA, June 1990. ACM Press.

[RGAB98] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz Andre Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, oct 1998.