Parallelization of Utility Programs Based on Behavior Phase Analysis

Xipeng Shen Chen Ding
{xshen,cding}@cs.rochester.edu

Computer Science Department, University of Rochester,
Rochester, NY, USA 14627

Abstract. With the fast development of multi-core processors, automatic parallelization becomes increasingly important. In this work, we focus on the parallelization of utility programs, a class of commonly used applications including compilers, transcoding utilities, file compressions, and databases. They take a series of requests as inputs and serve them one by one. Their high input dependence poses a challenge to parallelization.
We use active profiling to find behavior phase boundaries and then automatically detect run-time dependences through profiling. Using a unified framework, we manually parallelize programs at phase boundaries. We show that for two programs, the technique enables parallelization at large granularity, which may span many loops and subroutines. The parallelized programs show significant speedup on multi-processor machines.

1 Introduction

Nowadays due to the increasing complexity, it is difficult to improve the speed of high-performance uniprocessors. Chip multiprocessors is becoming the key of the next generation personal computers. But many applications, especially those running on past personal computers, are sequential programs and require parallelization to benefit from multiple cores.

In this paper, we describe a novel coarse-grain parallelization technique focused on a class of commonly used programs. *Utility programs* are a class of dynamic programs whose behavior strongly depends on their input. The examples include compilers, interpreters, compressions, transcoding utilities and databases. The applications all provide some sort of service: they accept, or can be configured to accept, a sequence of requests, and each request is processed more-or-less independently of the others. Because their behavior depend heavily on the input, utility applications display much less regular behavior than typical scientific programs. Many of them invoke many recursive function calls.

Our parallelization is based on behavior-based phase analysis. Here we define behavior as the operations of a program, which changes from input to input. A *behavior phase* is a unit of the recurring behavior in any execution. It may have plenty of loops and function calls. We use active profiling and pattern recognition techniques to detect phases [7]. Each instance of the top-level phase is the processing of a request. For example, the compilation of a function is a phase instance in GCC, and the parsing

of a sentence is a phase instance in Parser. The key observation is that the phases in utility programs coincide with program service periods and thus its *memory usage periods*. Operations inside a phase instance may have complex dependences but different phase instances are usually independent or can be made independent. Therefore, phase boundaries are good places to parallelize utility programs.

Phase-level parallelization has three steps. First, we automatically employ behavior-based phase analysis to find phases and mark them in the program source code through debugging tools [7]. Secondly, we discover the run-time data dependences among phase instances through profiling. Finally, we parallelize the program under a unified framework. The last step is currently semi-automatic, and the correctness requires the verification of a programmer. The step can potentially be automated with run-time system support, which automatically detects dependence violations and recovers when necessary. We have applied phase-level parallelization on two programs, Gzip and Parser and obtained up to 12.6 times speedup on a 16-processor machine.

There have been many efforts on automatic parallelization. Dynamic parallelization was pioneered more than a decade ago [5, 10]. Many recent papers studied thread-level speculation with hardware support ([2] contains a classification of various schemes). Most of those methods exploit loop-level parallelism, especially in low-level loops. Ortega et al. manually find parallelism in coarse loops for SPECInt benchmarks, which they call distant parallelism [4]. Our technique exploits parallelism at phase granularity for utility programs with some user support. It is orthogonal to fine-grain parallelism techniques.

2 Parallelization Techniques

2.1 Phase Detection

Phase detection is to find the boundaries of recurring behavior patterns in a program. We use a two-step technique to detect phases in utility programs. Active profiling uses a sequence of identical requests to induce behavior that is both representative of normal usage and sufficiently regular to identify outermost phases. It then uses different real requests to capture common sub-phases and to verify the representativeness of the constructed input. The phase boundaries are determined through statistic analysis on the dynamic basic block trace of the execution and are marked in the program code (see [7] for details.)

Phases have a hierarchical structure. In this study, we make use of the outermost phases only. We use process-based parallelization, where each phase instance is executed by a child process. If phase instances are small, a process can execute a group of phase instances at a time.

2.2 Phase-dependence Detection

Phase-dependence detection is to find run-time data dependences between phase instances during profiling runs. Similar to loop-dependence analysis[1], there are three kinds of phase-dependences: *flow dependence*, *antidependence*, and *output dependence*.

A flow dependence happens when a phase instance stores a value into a memory location and a later phase instance reads the value. An antidependence happens when a phase instance reads from and a later one writes to the same location. Finally, an output dependence happens when two phase instances write to the same memory location. Since process-based parallelization creates a separate address space for different phase instances, we can ignore anti- and output dependences. In comparison, thread-based parallelization must deal with them explicitly.

```
NODE* xlevel (NODE * expr) { /* function definition */
    if (++xltrace < TDEPTH){...} /* read and write xltrace */
    - -xltrace;} /* read and write xltrace */
```
(a) False dependence due to implicit initialization

```
char*buf;
...
buf[i] = 0; /* both load and store operations due to byte operations*/
```
(b) Addressing false dependence

```
*u = *t; /* load value of *t */
...
("t" is freed and "deletable" is allocated)
deletable[i][j] = FALSE; /* store operation*/
(deletable is freed)
...
```
(c) Allocator false dependence

```
for (sym = getelement(array,i);...){... } /* load "array" element*/
setelement (array, i, sym); /* store "array" element */
```
(d) Real phase dependence

Fig. 1. Example code of LI and Parser from the SPEC CPU2000 benchmark suite showing removable flow dependences. Each case contains a load from and a store to the same memory location. The dependence flows from the store in each phase instance to the load of the next phase instance.

Most flow dependences among phases are removable. Here "removable" means that the flow dependence can be safely ignored in process-based parallelization. We divide the removable flow dependences into three classes, give examples for each class, and describes the detection of these dependences as well as possible run-time support to guarantee correctness. As examples, Figure 1 shows fragments of the code of LI and Parser in the SPEC CPU2000 benchmark suite. Figure 1 (a) shows the first class of removable dependence, which are caused by *implicit initialization*. In this case, the variables are reset to their initial value at the end of each phase instance. In the example, the global variable *xltrace* is a stack pointer. It increments by 1 when evaluating an expression and the evaluation function may call itself to evaluate subexpressions. It decrements by 1 after the evaluation. It is always equal to -1, its initial value, at the beginning and the end of each phase instance. Such objects can be automatically detected by checking the consistency of their values at the beginning of every phase. If their

values are always the same, the variables are likely to belong to this class. In the runtime system, the value of those variables may be checked dynamically, and the parallel process may be rolled back when the value is unexpected.

Figure 1 (b) shows the second class of removable dependences, *addressing dependences*. In this class, the source code has no loading operations but the binary code has because of addressing. In the example code, there is a write of a byte. The code is compiled into the following assembly code on a Digital Alpha machine, which does not have a byte write operation.

```
lda     s4, -28416(gp) /* load array base address */
addq    s4, s0, s4     /* shift to the target array element */
ldq_u   v0, 0(s4)      /* load a quadword from the current element */
mskbl v0, s4, v0       /* set the target byte to 0 by masking */
stq_u   v0, 0(s4)      /* store the new quadword to the array */
```

Our dependence detection finds a flow dependence between instruction "stq_u v0, 0(s4)" and "ldq_u v0,0(s4)" when more than one phase instances execute that statement. It is removable dependence since the loading operation is purely for the store operation and the value of the loaded location has no effects on the program's execution.

Finally, flow dependences may happen when memory is reused across phase instances by a dynamic memory allocator. Figure 1 (c) shows an example. The two objects $*t$ and *deletable* are independent and have exclusive live periods. They are allocated to the same memory region by the memory allocator. Process-based parallelization can ignore all these three classes of flow dependences because it uses separate address spaces for processes.

Figure 1 (d) shows a real phase-dependence. An earlier phase instance fills *array* with some calculation results, which are loaded in a later phase instance. For correctness, the parallelization must protect such objects to avoid the violation of the dependence.

We develop an automatic tool to trace memory accesses in profiling runs, detect different kinds of dependences, and then find the corresponding source code. The tool cannot detect all possible dependences in a program but it can help the parallelization process by finding the likely parallel regions in the program. The tool first instruments the binary code of a program to monitor memory accesses for dependence detection through instrumentor ATOM [8] and then finds and displays the source code related to phase-level flow dependences that are not removable. Trace-level dependence tracking has been used extensively for studying the limit of parallelism (an early example is give by Kumar [3]) and for easing the job of debugging.

The effect of profiling depends on the coverage, that is, the portion of phase-level dependences that we find through profiling. Multiple training runs help to improve the coverage. Utility programs take a series of requests as an input; thus one execution contains the process of many different requests. This leads to good coverage even with a single input.

2.3 Program Transformation

Our parallelization is process based. Thread-based parallelization is an alternative. Processes have their own address space and are thus more independent than threads. Utility programs often have just a small number of phase-level flow dependences, since phase instances coincide with the memory usage period of a program. For example, variable *optind* is the only flow dependence that is not removable in benchmark Gzip and requires code movement. There are other global data structures shared by phase instances, but they can be privatized since they introduce no flow dependences. In thread-based parallelization, multiple threads share the address space and must deal with antidependences and output dependences explicitly. In our study, we use process-based parallelization.

Many utility programs have a common high level structure; they first read requests and then process them one by one. We design a unified framework for their parallelization, which is shown in our technical report [6] for lack of space.

There are two strategies for parallelization. One is to let programmers know the phase structure and the detected dependences so the programmer may parallelize the program when possible. The manual effort is simplified by the automatic tools, which suggest parallel regions that have good efficiency and scalability. The other strategy is run-time dependence detection and phase-level speculation. All objects are put into a protected region so speculation can be canceled when it fails. To improve efficiency and scalability, we can give special treatment to the dependences detected in the profiling runs. The operating system can monitor the accesses to those objects and roll back later processes only when the dependences on those objects are violated or an earlier process writes to the other objects in the protected region. This strategy requires less manual work but requires the support of operating system. It has extra overhead. We use the first strategy in this work and are in the process of developing the second strategy.

3 Evaluation

We apply phase-level parallelization on Parser and Gzip in Spec CPU2000 suite. We first discuss the issues in the parallelization of each benchmark, then report the performance of the parallelized programs.

3.1 Parser

Parser is a natural language (English) parsing program. It takes a series of sentences as its input and parses them one by one. The detected phase boundary is before the parsing of a sentence. The dependence detector reports 208 dependences in which 6 are distinct flow dependences. Figure 2 shows part of the report. Among the six, the first two are removable addressing dependences. The next two are implicit initialization removable dependences: the store statement for *lookup_list* traverses the list to free the list elements and *mn_free_list* is freed at the end of each phase instance. The rest two dependences are not easily removable. They include four global variables as array *user_variable*, *unknown_word_defined*, *use_unknown_word* and *echo_on*. The last three variables are

elements of array *user_variable*. They are a set of global boolean variables, used to configure the parsing environment. For example, variable *echo_on* determines whether to output the original sentence or not. As the program comes across a command sentence like "!echo" in the input file, it turns on the boolean value of *echo_on*. The command sentences are part of the input file. In our parallelization, we let the root process handle those command sentences first and then create child processes to parse every other sentence with the corresponding environment configuration.

Dep. type	Operation	File	Line	Source code
addressing	store	analyze-linkage.c	659	patch_array[i].used = FALSE;
	load	analyze-linkage.c	659	patch_array[i].used = FALSE;
addressing	store	and.c	540	if (*s == '*') *u = *t;
	load	and.c	540	if (*s == '*') *u = *t;
initialization	store	read-dict.c	763	lookup_list = n;
	load	read-dict.c	760	while(lookup_list != NULL) {
initialization	load	fast-match.c	59	mn_free_list = m;
	store	fast-match.c	44	if (mn_free_list != NULL) {
real dep.	store	main.c	620	(∗(user_variable[i].p)) = !!(∗(user_variable[i].p));
	load	main.c	1557	if (!(unknown_word_defined && use_unknown_word)) {
real dep.	store	main.c	674	∗(user_variable[j].p) = !(∗(user_variable[j].p));
	load	main.c	1573	if (echo_on) printf("%c ", mc);

Fig. 2. The partial report of dependences in Parser. Each pair of rows show the two lines of code causing a phase-level flow dependence.

3.2 Gzip

Gzip is a popular data compression program (the Gzip used in the experiment is configured as the GNU compression program instead of the SPEC CPU2000 default version). It takes a series of files as the input and compresses them one by one. It has only one phase-level flow dependence, variable *optind*, which counts the number of input files. It increases by one in each phase instance. Automatically our analysis tool finds the boundary of file processing as the phase boundary. The dependence detection finds the flow dependence. After applying the unified framework for transformation, the only remaining work is to move the counter increment operation from child processes to the root process to resolve the dependence.

3.3 Methodology

We measure the performance on two multi-processor machines, whose configurations are shown in Table 1.

We use "gcc -O3" for compilation. The Gzip has only one input file as its test input. We duplicate the file multiple times as the regular input for profiling. The ref input of Gzip contains too few files to measure the effectiveness. We create 105 files by duplicating all ref inputs 20 times each. The regular input for Parser is a file containing six identical English sentences. We use the test input for the dependence profiling of Parser and the ref input for evaluation.

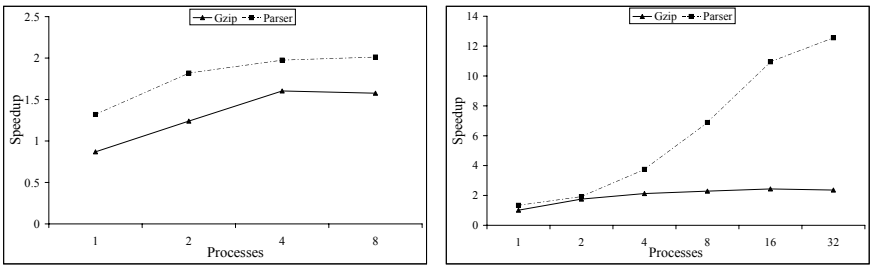
Table 1. Machine configurations

CPU Number	4	16
CPU Type	Intel Xeon	Sunfire Sparc V9
CPU Speed	2.0GHz	1.2GHz
L1 cache	512K	64K

3.4 Speedup

Figure 3 (a) and (b) show the speedup curves of the parallelized programs on Intel Xeon and Sunfire multi-processor machines. The x-axis is logarithmic. We experiment up to 8 processes on the 4-CPU Intel machine. When the number of process is smaller than the number of processors, the speedup increases significantly and reaches the peak when the process number is equal to the processor number. Gzip achieves 1.6 times speedup and Parser achieves 2.0.

On the 16-CPU Sunfire machine, Parser shows 12.6 times speedup with 16 processes. Gzip shows 2.4 times speedup. The limited speedup of Gzip is due to the saturation of file I/O.



(a) Speedup on an Intel Xeon machine (b) Speedup on a Sunfire Sparc V9 machine

Fig. 3. Speedup of parallelized programs on multiple-processor machines

4 Related Work

There have been many efforts on automatic parallelization. The most related work is the study from software aspects. Those efforts roughly include two classes: task and data based parallelization.

Previous parallelization techniques are based on static compiler or run-time techniques. The former tries to find the parallelism opportunities in program loops through static dependence analysis in compilers and then applies privatization and reduction parallelization [1, 9].) Static techniques work well for programs with regular, statically analyzable access patterns. Run-time parallelization can reveal and exploit input dependent and dynamic parallelism in loop nests [5, 10]. In comparison, this work uses

profiling to parallelize programs at phase boundaries and exploit parallelism that may span many loops and subroutines.

Another class of parallelization is based on Thread-level Speculation (TLS) hardware. TLS hardware provides support for speculative threads and dynamical roll back given the violation of dependences. This work partially relies on programmer knowledge but requires no special hardware support.

5 Conclusions

In this work, we propose a phase-level parallelization technique for parallelizing utility programs. It finds phase boundaries by active profiling, identifies common phase-level dependences by training, handles anti- and output dependences through process-based parallelization, classifies removable flow dependences, and relies on programmer support to handle remaining flow dependences and ensure the correctness of the transformation. Unlike previous work, this technique helps to parallelize a program in coarse granularity, which may span many loops and subroutines. Our preliminary experiments show significant speedups for two non-trivial utility programs on multi-processor machines.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.
2. M. J. Garzaran, M. Prvulovic, J. M. Llaberia, V. Vinals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *Proceedings of International Symposium on High-Performance Computer Architecture*, 2003.
3. M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37, 1988.
4. D. Ortega, Ivan Martel, Eduard Ayguade, Mateo Valero, and Venkata Krishnan. Quantifying the benefits of specint distant parallelism in simultaneous multi-threading architectures. In *Proceeding of the Eighth International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, California, October 1999.
5. L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.
6. X. Shen and C. Ding. Parallelization of utility programs based on behavior phase analysis. Technical Report TR 876, Department of Computer Science, University of Rochester, September 2005.
7. X. Shen, C. Ding, S. Dwarkadas, and M. L. Scott. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.
8. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994.
9. M. Wolfe. *Optimizing Compilers for Supercomputers*. The MIT Press, 1989.
10. C. Q. Zhu and P. C. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, 13(6), 1987.