# A Runtime Optimization System for OpenMP[⋆]

Mihai Burcea and Michael J. Voss

(burceam,voss)@eecg.toronto.edu
Edwards S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto, Toronto, ON, CANADA

**Abstract.** This paper introduces stOMP: a specializing thread-library for OpenMP. Using a combined compile-time and run-time system, stOMP specializes OpenMP parallel regions for frequently-seen values and the configuration of the runtime system. An overview of stOMP is presented as well as motivation for the runtime optimization of OpenMP applications. The overheads incurred by a prototype implementation of stOMP are evaluated on three Spec OpenMP Benchmarks and the EPCC scheduling microbenchmark. The results are encouraging and suggest that there is a large potential for improvement by the runtime optimization of OpenMP applications.

## 1 Introduction

In recent years, there has been an increased interest in the runtime optimization of applications. At compile-time, optimizers are severely restricted by incomplete knowledge of the program input, target architecture, and library modules. By performing optimization at runtime, more complete knowledge is available allowing programs to be highly tuned to their exact runtime environment and usage.

In this paper, we present stOMP: a specializing thread-library for OpenMP. stOMP is an implementation of the OpenMP API [1, 2] based on the Omni compiler and runtime library [3]. In addition to the implementation of OpenMP provided by Omni, stOMP leverages properties of OpenMP applications and the Omni runtime library to provide an environment for the dynamic optimization and compilation of parallel regions. At runtime, stOMP is used to *specialize* parallel regions for runtime constant or well-behaved variables, and for the current configuration of the parallel environment.

In Section 2, we present related work in dynamic and adaptive program optimization. In Section 3, we discuss the unique properties of OpenMP applications that make them ideal for runtime optimization. In Section 4, we present an overview of the stOMP compilation and runtime subsystems, and details of the stOMP prototype that is currently under development. An initial evaluation of our prototype is provided in Section 5. In Section 6, we present our conclusions and plans for future work.

[⋆] This work is supported in part by the National Science and Engineering Research Council, Bell Canada University Labs, the Connaught Foundation and Dell Canada.

## 2 Related Work in Runtime Optimization

Runtime compilation and optimization has been most successfully applied in languages that have high abstraction penalties, such as Java, C# and Smalltalk [4–7]. Often in these languages, runtime compilation can be used to remove some of the high overheads incurred by advanced language features, such as virtual/polymorphic function invocation, dynamic class loading and array bounds checking. Just-in-time and adaptive compilation are now accepted practice for these languages.

There have been mixed results for dynamic optimization when applied to non-object-oriented imperative languages, such as C. The DyC, 'C and Tempo projects have explored user-directed dynamic compilation of C programs [8–10]. These approaches require user-annotation of application programs to select code regions and variables for optimization. Unlike Java, C# and Smalltalk, C programs are less tolerant of the overheads incurred by runtime compilation, and therefore users must carefully select code regions where dynamic compilation will be profitable. In [11], a profile-directed tool, Calpa, is used to automate the selection of regions and specialization targets for DyC. However in [11], it is unclear if the approach scales to large programs.

In [12], ADAPT is proposed for the runtime optimization of loop-based Fortran applications. ADAPT is a generic system that allows compiler developers to easily add and experiment with adaptive optimizations. In [12], the dynamic serialization of parallel OpenMP regions is used as an example application of the system. Unlike stOMP, ADAPT cannot directly influence or leverage the OpenMP runtime library.

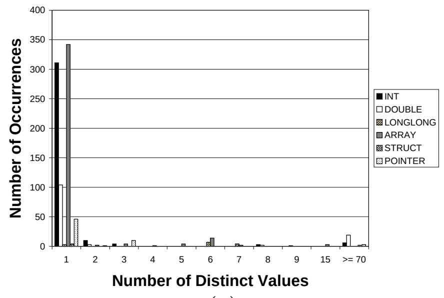
## 3 OpenMP as a Target for Runtime Optimization

The OpenMP API has bindings for both the Fortran [1] and C/C++ [2] languages. As discussed in Section 2, dynamic compilation systems that target these languages usually require users to hand-annotate regions for optimization [8–10]. However in the case of OpenMP applications, we believe that useful optimization directives can be inferred from the already existing OpenMP directives, and that OpenMP libraries provide needed features for easily implementing a runtime optimization system.

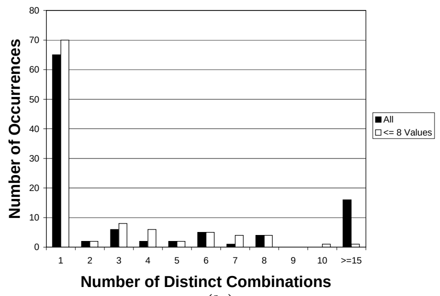
### 3.1 Runtime Characteristics of OpenMP Applications

To perform runtime optimization on an application, two basic choices must be made: (1) what code regions should be optimized and (2) what runtime values should be used to specialize these regions. A natural choice for the code regions to optimize in OpenMP applications are PARALLEL regions. These regions have been annotated by users because they are important to the application's performance and they have execution times that are large enough to tolerate parallelization overheads. These traits make them not only ideal for parallelization but for runtime optimization as well.

The second choice that needs to be made is to select the runtime values to use in optimization. Fig. 1 shows the behavior of shared variables in the C and Fortran77 SPEC OpenMP benchmarks [13]. In Fig. 1 (a), a histogram of the number of distinct values per shared-variable is shown by type. Fig. 1 (a) demonstrates that shared variables have only a few values over a program run.



(a)



(b)

**Fig. 1.** Figure (a) is a histogram of the number of distinct values held by shared variables at entry to parallel regions in the C and Fortran77 SPEC OpenMP Benchmarks. For ARRAYs, STRUCTs and POINTERs, "Values" correspond to the addresses of the variables, not their content. Figure (b) is a histogram of the total number of combinations of variable values seen at entry to parallel regions. In (b), the combinations obtained when using all variables, as well as the well-behaving variables with less than 8 distinct values each are shown. In both (a) and (b), "Occurrences" is the static count of regions or variables that exhibit the behavior.

Fig. 1 (b) shows the number of distinct combinations of variable values. A combination of variable values is the set of values held by all variables of interest at entry to the region. For example, if a region has two shared variables, A and B, a combination might be $A = 1, B = 3$. In Fig. 1 (b), we present data for combinations derived from all shared variables, as well as combinations derived from the shared variables that have 8 or less distinct values (as shown in Fig. 1 (a)). Fig. 1 (b) suggests that if regions are optimized for frequently-seen combinations of values, only a small number of code versions need to be compiled and managed at runtime.

## 3.2 Features of OpenMP Implementations

Standard implementations of OpenMP provide the necessary hooks for easily implementing a runtime optimization system. Fig. 2 shows an example of an OpenMP parallel region as annotated by a user. In Fig. 3 (a) and 3 (b), the code generated by the Omni compiler to implement this region is shown.

The call to `_ompc_do_parallel` in Fig. 3 (a), causes the Omni runtime library to create a team of threads to execute the code found in the `__ompc_func_0` routine. In Section 4, we propose a system that uses the call to `_ompc_do_parallel`

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
  iam = omp_get_thread_num();
  np = omp_get_num_threads();
  ipoints = npoints / np;
  subdomain(x, iam, ipoints);
}
```

**Fig. 2.** The source code for a simple parallel region.

to select and create highly specialized and optimized versions of code for each parallel region at runtime.

### 3.3 Optimization Opportunities

If shared variables can be used to specialize OpenMP applications, there are a number of opportunities for enhancing the performance of these applications. An inspection of the SPEC OpenMP benchmarks shows that simple expressions of shared variables, the thread id and the number of threads, often determine loop bounds and branch conditions. Many static optimizations are severely handicapped when loop bounds and conditionals are unknown. By capturing these values at runtime, more accurate and aggressive loop transformations might be applied to these important parallel nests.

In addition, the transformations applied by OpenMP compilers often create complicated code that is less amenable to compiler optimization. For example, functions calls are added to determine loop schedules, and shared variables may be accessed through pointers, instead of directly, leading to added aliases. A runtime optimization system may be able to see through, or remove, these complications using runtime information, generating better code as a result.

## 4 The stOMP Runtime Optimization System

Fig. 4 shows an overview of our proposed system, stOMP. The system consists of three major components: a modified version of the Omni OpenMP compiler, a modified version of the Omni runtime library and a Dynamic Optimizer. We briefly describe these three components in the sections below.

### 4.1 The stOMP Compiler

For each parallel region in the OpenMP application, the stOMP compiler creates a new file that contains the source code for that region. For the code in Fig. 2, this new file would contain a copy of the code shown in Fig. 3 (b). Special attention is paid to ensure that global variables, static variables, and Fortran Common blocks are properly transformed to allow the code to be placed in a separate file, while retaining the semantics of the original code. These files are stored for

```
(*(__ompc_argv)) =
  (((void *) (&x)));

(*((__ompc_argv) + (1))) =
  (((void *) (&npoints)));

_ompc_do_parallel (__ompc_func_0,
                   __ompc_argv);
```

(a)

```
static void
__ompc_func_0 (__ompc_args)
    void **__ompc_args;
{
  auto int _p_iam;
  auto int _p_np;
  auto int _p_ipoints;
  auto int **_pp_x = (((int **)
           (*__ompc_args)));
  auto int *_pp_npoints = (((int *)
    (*((__ompc_args) + (1)))));
  (_p_iam) = (omp_get_thread_num());
  (_p_np) = (omp_get_num_threads());
  (_p_ipoints) =
     ((*_pp_npoints) / (_p_np));
  subdomain (*_pp_x, _p_iam,
             _p_ipoints);
}
```

(b)

**Fig. 3.** The parallel region from Fig. 2 after being transformed by Omni: (a) the site of the parallel region and (b) the function created by Omni to encapsulate the body of the parallel region.

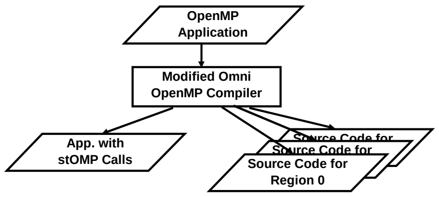
future processing by the stOMP Dynamic Optimizer. In addition to generating the parallel region files, the stOMP compiler also creates application-specific functions for runtime code management and compilation.

## 4.2 The stOMP Runtime Library

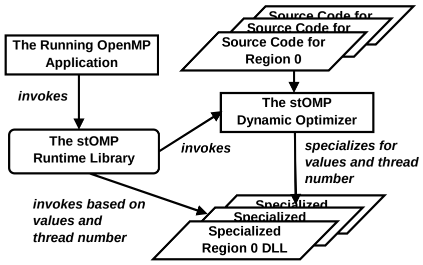
The runtime system, shown in Fig. 4 (b), tracks shared variables and invokes the stOMP Dynamic Optimizer. The runtime library maintains 8-element, per-region code caches that store dynamically generated code. At each call to the `_ompc_do_parallel` function, the code cache corresponding to the first argument is inspected for a version that matches the current values of the shared variables, number of threads and thread id (referred to as a *combination* of values in Section 3).

Fig. 5 shows an overview of the flow through the `_ompc_do_parallel` function. Before spawning worker threads, the master thread invokes the match function generated by the stOMP compiler for this parallel region. The match function returns a set of functions, one for each thread, that should be invoked to execute the parallel region. If this region has already been specialized for the current combination of runtime values, the set of specialized functions is returned.

If no matching functions are found, the number of times this combination of values has been seen for this region is compared against a user-set threshold.



(a)


The Running OpenMP Application
invokes
The stOMP Runtime Library
invokes
Source Code for
Source Code for
Source Code for Region 0
The stOMP Dynamic Optimizer
specializes for values and thread number
invokes based on values and thread number
Specialized
Specialized
Specialized Region 0 DLL

(b)

**Fig. 4.** An overview of stOMP: (a) the compiler and (b) the runtime system.

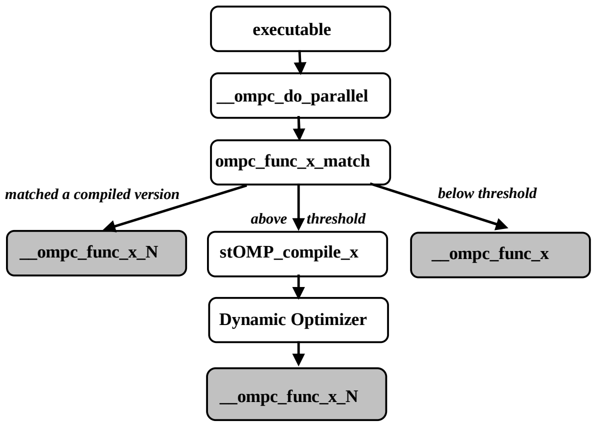


**Fig. 5.** The process used by the match function to select a copy of the parallel region to execute: The gray boxes represent the selected versions. The __ompc_func_x box is the statically-compiled default version of the code and the __ompc_func_x_N box is a region specialized for the current shared variables, number of threads and thread id.

If the threshold has been exceeded, the stOMP_compile function for this region is invoked to create the corresponding set of functions. If the threshold has not yet been exceeded, the statically-compiled default set of functions is returned and a counter for this combination of values is incremented. If more than 8 combinations are seen for a region, the least-frequently-used cache block will be evicted.

In addition to the heuristic in Fig. 5, we also always specialize immediately for the combination seen during the first invocation of each parallel region. We immediately specialize for the first combination since Fig. 1 shows that most regions have only a single combination of values. This approach also allows regions that execute only once to benefit from runtime optimization.

## 4.3 The stOMP Dynamic Optimizer

The stOMP Dynamic Optimizer is a hand-coded preprocessor that calls gcc as a back-end to generate shared libraries. The `stOMP_compile_x` function, shown in Fig. 5, invokes the Dynamic Optimizer and dynamically loads the resulting libraries into the executable using standard `dlopen` and `dlsym` calls. When the stOMP_compile function is invoked by the runtime library, it is passed the current number of threads and the current values (or addresses) of the shared variables in use by the region.

A single call to the Dynamic Optimizer creates P specialized versions, one for each thread. The Dynamic Optimizer is coded in C and uses OpenMP pragmas to perform compilation of these P versions in parallel. However, the Dynamic Optimizer itself is currently not being optimized by stOMP. We have yet to fully explore mechanisms for decreasing the overhead of our runtime compilation system, but believe that significant reductions are possible.

Our current optimizer implements a very limited form of runtime constant propagation. The values or addresses of shared variables are explicitly assigned at the top of each parallel region function, providing more accurate values and addresses to the back-end optimizer. Also, each call to `omp_get_thread_num` and `omp_get_num_threads` is replaced by its corresponding runtime value (as communicated by the runtime library). Fig. 6 shows an example of a snippet from a parallel region in Equake, a SPEC OpenMP benchmark, before and after optimization by our current Dynamic Optimizer.

```
extern struct smallarray_s **w1;
extern int ARCHnodes;

void quake__ompc_func_31 (__ompc_args)
     void **__ompc_args;
{
  auto int *_pp_j;
  (_pp_j) = (((int *) (*__ompc_args)));
  {
    auto int _p_i;
    auto int _p_i_28;
    auto int _p_i_29;
    auto int _p_i_30;
    (_p_i_28) = (0);
    (_p_i_29) = (ARCHnodes);
    (_p_i_30) = (1);
    _ompc_default_sched (&_p_i_28,
                &_p_i_29, &_p_i_30);
```

```
extern struct smallarray_s **w1;
extern int ARCHnodes;

void
quake__ompc_func_31 (__ompc_args)
     void **__ompc_args;
{
  int _stomp_pp_j = (int) 0;
  auto int *_pp_j;
  _pp_j = &_stomp_pp_j;
  w1 = 0x8538d30;
  ARCHnodes = 30169;
  {
    auto int _p_i;
    auto int _p_i_28;
    auto int _p_i_29;
    auto int _p_i_30;
    (_p_i_28) = (0);
    (_p_i_29) = (ARCHnodes);
    (_p_i_30) = (1);
    _ompc_default_sched (&_p_i_28,
                &_p_i_29, &_p_i_30);
```

**Fig. 6.** An example of code from one of the parallel loop in Equake: (a) the code before being transformed by the runtime optimizer and (b) the code after runtime optimization.

The optimizations currently implemented in our Dynamic Optimizer, as Shown in Fig. 6, are unlikely to yield large performance improvements on most programs. However, our current prototype does allow us to gain invaluable insight into the lower-bound on the overheads we can expect from our full system. A study of these overheads is presented in the next section.

## 5 A Preliminary Evaluation

As described in Section 4, only limited optimizations have been implemented in the current stOMP prototype. In this Section, we therefore present an initial study of the overheads associated with our system. We first explore the overheads added to parallel regions by examining the EPCC Scheduling Microbenchmark (`schedbench`) [14]. Next, we show results from applying our system to three of the SPEC OpenMP Benchmarks: Apsi, Art and Equake.

Our test system is a 4-processor Intel Xeon server with 1.6 GHz Hyper-threaded processors running Redhat Linux 7.3. We present results for 1 through 4 threads, restricting ourselves to the number of physical processors in the system. All code is compiled using gcc version 2.96 with the `-O2` optimization flag.

### 5.1 The performance of the EPCC scheduling microbenchmark

Fig. 7 shows results from schedbench. This benchmark determines the overheads associated with starting and scheduling parallel regions. stOMP uses calls to `_ompc_do_parallel` to select and compile specialized code, and therefore its overheads are directly measured by this benchmark.

In Fig. 7, the improvement of `schedbench` when optimized by stOMP is presented. Fig. 7 (a) and (b) show the improvement for the static scheduling part of the microbenchmark. This benchmark executes a statically scheduled parallel region using varying chunk sizes (ranging from 1 to 128). The benchmark executes the region 50 times for each configuration, and calculates the average execution time of the region for each chunk size.

Fig. 7 (a) shows the improvement calculated from the average execution time measurements. These measurements include the overheads incurred by our Dynamic Optimizer. The Optimizer will be invoked once for each <processor number, chunk size> pair. Fig. 7 (b) shows the improvement calculated when the Dynamic Optimizer overhead is removed (this result includes 49 invocations of the parallel region).

Fig. 7 (b) suggests that the overhead incurred by the stOMP Dynamic Optimizer is the major reason for the poor performance shown in Fig. 7 (a). In fact, when the runtime compilation overhead is removed, stOMP outperforms the orginal Omni code in almost all cases, with an average improvement of 13.2% and gains as large as 33%.

The measurements shown in Fig. 7 (b) include all of the code cache lookup and management overheads incurred by the runtime library. These results show that improvements are possible from the simple optimizations currently implemented in stOMP, but that the runtime of our Optimizer might need to be reduced.

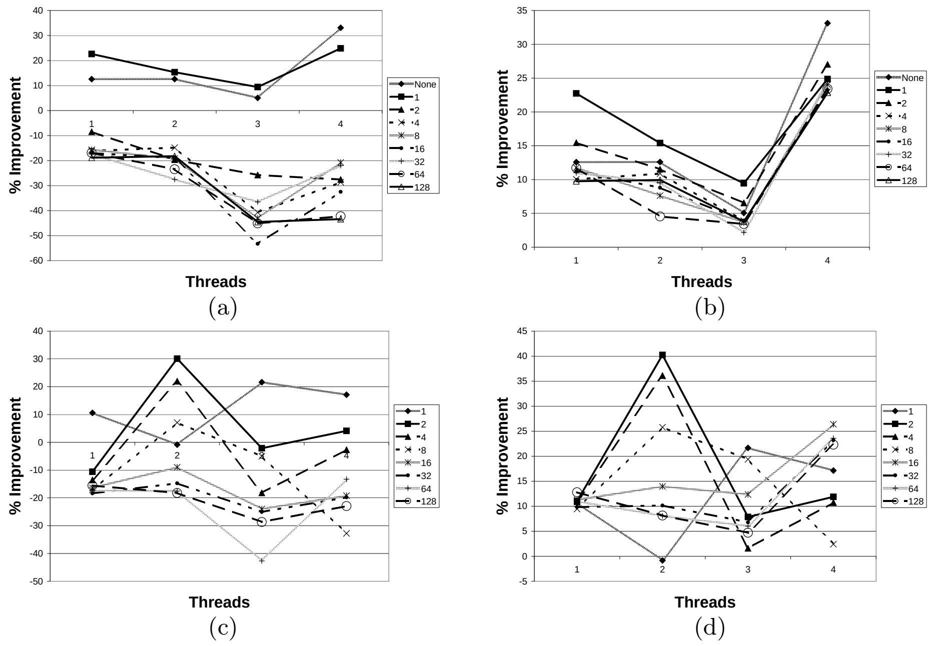


**Fig. 7.** Evaluation of the EPCC schedbench benchmark. The improvement of stOMP over Omni on the statically scheduled loop (a) including the time to spent in the Dynamic Optimizer and (b) excluding the time spent in the Dynamic Optimizer. The improvement of stOMP over Omni on the dynamically scheduled loop when (c) including the time spent in the Dynamic Optimizer and (d) excluding the time spent in the Dynamic Optimizer.

The results for the dynamic scheduling part of `schedbench` are shown in Fig. 7 (c) and 7 (d). These results reinforce the conclusions drawn from Fig. 7 (a) and 7 (b). When the Dynamic Optimizer overhead is ignored, the average improvement on the dynamic loop is 14%, with improvements as large as 40%.

## 5.2 The performance of Apsi, Art and Equake

Fig. 8, 9 and 10 show the performance of three Spec OpenMP Benchmarks: Apsi, Art and Equake. For both Apsi and Art we used the SPECOMP2001 train data set and for Equake we used the SPECOMP2001 reference data set. Due to the limited optimizations currently implemented in our system, both Apsi and Equake perform worse with stOMP than with the original Omni compiler and library.

*Apsi:* Apsi has the largest number of parallel regions (with 28) of the three benchmarks that we tested, and also has a large number of shared variables for each parallel region. All regions in Apsi have at least 10 shared variables, with some having as many as 18. We therefore expect to see a larger overhead with Apsi. Fig. 8 (a) shows that on 4 processors Apsi runs 10% slower than the original version. Fig. 8 (b) clearly indicates that this degradation is due to the time spent in the Dynamic Optimizer. Because of the large number of regions, the runtime optimizer is invoked frequently, generating 136 shared libraries during a single execution of the benchmark.

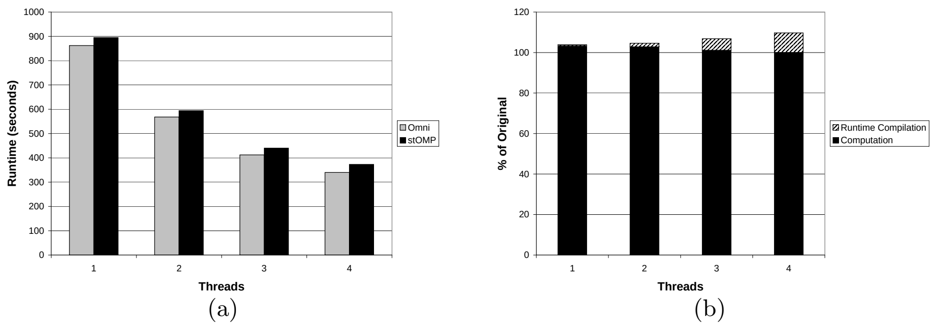
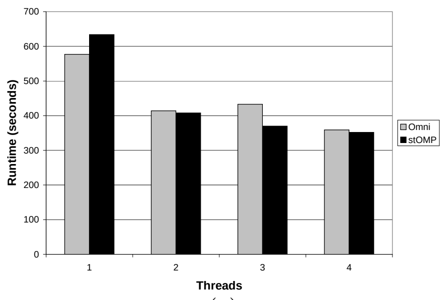


**Fig. 8.** The Spec OpenMP Benchmark Apsi: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version.

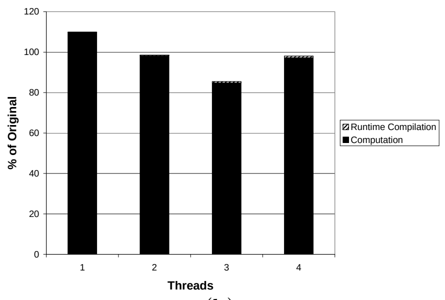
*Art:* Art has 3 parallel regions, each of which is only invoked once. Therefore stOMP just-in-time compiles each region once and incurs no further overheads from compiling or matching. Consequently, the performance of Art in Fig. 9 (a) and (b) is the best of the three benchmarks. The stOMP version of Art shows improvements for all but the 1-processor run, with a gain of 14% on 3 processors. The runtime compilation overheads for Art are negligible.

*Eqake:* Equake has 11 parallel regions, many of which are called multiple times. In Fig. 10 (a), the stOMP version of Equake is always within 3% of the original code. Fig. 10 (b) shows that while compilation overheads increase with the number of threads, the time spent in the Dynamic Optimizer is small.

The results from our initial evaluation of stOMP are encouraging. Both Art and Equake show small runtime compilation overheads, with Art already showing an improvement from the limited optimizations performed by our prototype. The overhead of stOMP when running Equake is always less than 3% and when running Apsi is always less than 10%. In the future, we plan to look at running the Dynamic Optimizer in the background, as is done in ADAPT [12], and thereby hide some of the latencies incurred in programs, such as Apsi, that have a large number of parallel regions.

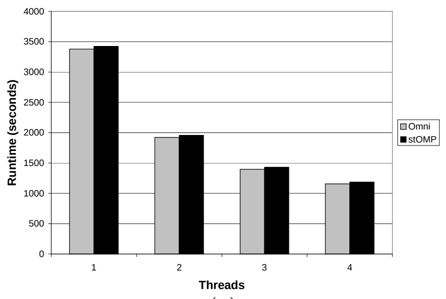


(a)

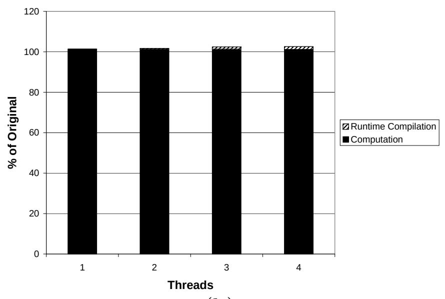


(b)

**Fig. 9.** The Spec OpenMP Benchmark Art: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version.



(a)



(b)

**Fig. 10.** The Spec OpenMP Benchmark Equake: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version.

## 6 Conclusions

In this paper, we have introduced stOMP: a specializing thread library for OpenMP. The stOMP system is built on the Omni OpenMP compiler and library, and provides a system for the runtime optimization of parallel regions. In Section 3, we present motivation for the runtime optimization of OpenMP applications. In the SPEC OpenMP Benchmarks, it is shown that shared variables are in general runtime constant, or have only a few values during a program's execution.

In Section 4, we describe the architecture of stOMP. Using a combined compile-time and run-time system, stOMP specializes parallel regions for frequently-seen values and the configuration of the runtime system. In Section 5, a preliminary evaluation of stOMP is presented. Our results are encouraging and suggest that there is room for improvement using runtime optimization, but that our runtime compilation system needs to be refined to minimize overheads. In future work, we will explore a range of runtime optimizations using the stOMP system.

## References

1. OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, V. 2.0*, 2002.
2. OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, V. 2.0*, 2002.
3. The Omni OpenMP Compiler. http://phase.etl.go.jp/Omni/, 2003.
4. Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical White Paper, http://java.sun.com/products/hotspot/whitepaper.html, April 1999.
5. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proc. of the ACM SIGPLAN 2000 Conf. on Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, MN, October 2000.
6. Standard ECMA-335: Common Language Infrastructure (CLI). http://www.ecma.ch/ecma1/STAND/ecma-335.htm, February 2002.
7. L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. of the Conf. on Principles of Programming Languages*, Salt Lake City, Utah, 1984.
8. Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 293–304, Atlanta, GA, May 1999.
9. Massimiliano Polettto, Wilson C Hsieh, Dawson R Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
10. Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In *Proc. of the SIGPLAN '99 Conf. on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.
11. Markus Mock, Craig Chambers, and Susan Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *33rd Annual Symposium on Microarchitecture*, December 2000.
12. Michael Voss and Rudolf Eigenmann. High-Level Adaptive Program Optimization with ADAPT. In *Proc. of PPoPP'01: Principles and Practice of Parallel Programming*, Snow Bird, Utah, June 2001.
13. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools (WOMPAT)*, pages 1–10, July 2001.
14. J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *European Workshop on OpenMP (EWOMP)*, 1999.