To Inline or Not to Inline? Enhanced Inlining Decisions

Peng Zhao, José Nelson Amaral

Department of Computing Science
University of Alberta, Edmonton, Canada
{pengzhao, amaral}@cs.ualberta.ca

***Abstract*—**
The decision to inline a procedure in the Open Research Compiler (ORC) was based on a *temperature* heuristics that takes into consideration the time spent in a procedure and the size of the procedure. In this paper we describe the trade-off that has to be worked out to make the correct inlining decisions. We introduce two new heuristics to enhance the ORC inlining heuristics: *adaptation* and *cycle_density*. With *adaptation* we are allowed to vary the temperature threshold and prevent penalizing small benchmarks. With *cycle_density* we prevent the inlining of procedures that have a high temperature in spite of being called infrequently. We implemented our improved heuristics in the ORC and tested them for the Intel® Itanium™ Processor Family Platform using the SPEC INT2000 benchmark suite. While adaptation improves the speedup obtained with inlining across the SPEC2000 suite, *cycle_density* reduces significantly both the code growth and compilation time increase caused by inlining. We then characterize the SPEC INT2000 benchmarks according to the inlining potential of their function calls. Our enhancement is released in the ORC 2.0.

I. Introduction

Function inlining is a very important optimization technique that replaces a function call with the body of the function [2], [5], [6], [7], [8], [10], [13], [19], [14]. One advantage of inlining is that it eliminates the overhead resulting from function calls. The savings are especially pronounced for applications where only a few call sites are responsible for the bulk of the function invocations because inlining those call sites significantly reduces the function invocation overhead. For example, MCF (one of the SPEC2000 benchmarks) contains 34 call sites. Among these call sites, there are 5 that are executed more than 10 million times and 4 call sites that are executed more than 1 million times in a standard SPEC2000 training execution. These 9 call sites account for 99.85% of all the function invocations in MCF. Our experiments show that inlining the 15 most frequent call sites can reduce the running time of MCF by more than 9%.

Inlining also expands the context of static analysis. This wider scoped analysis creates opportunities for other optimizations. Because the body of the callee is now available at the call site, conservative assumptions that the compiler would previously make about the call site are no longer required.

Another advantage of inlining is the improvement of cache efficiency. From the point of view of the data cache (D-cache), after inlining the caller's variables that are referenced by the callee do not need to be passed as parameters. Thus, a variable that previously had separate representations in the caller and in the callee can now be reduced to a single memory location or even promoted to a register. This storage consolidation reduces the data access footprint of the application and improves the use of the memory hierarchy. A similar advantage also exists for the instruction cache (I-cache). After inlining, closely related segments of code are placed together, reducing chances of instruction cache conflicts [16].

However, inlining has negative effects. One problem with inlining is the growth of the code, also known as *code bloat*. Because a procedure may be called from multiple call sites, It is often impossible to eliminate a procedure after inlining a single call site. Thus, the final executable file must contain several copies of the procedure: the original one (if it is not eliminated as a dead function) and the inlined copies.

With the growth of functions because of inlining, the compilation time and the memory space consumption may become intolerable because some of the algorithms used for static analysis have super-linear complexity.

Besides the time and memory resource cost, inlining might also have the adverse effect of increasing the execution time of the application. After inlining the register pressure may become a limitation because the caller now contains more code, more variables, and more intermediate values. This additional storage requirement may not fit in the register set available in the machine. Thus, inlining may increase the number of register spills resulting in a larger number of load and store instructions executed at runtime.

Moreover, because the caller becomes larger after inlining, the possibility that it will either have instruction cache conflicts among its own instructions or interfere with other procedures in the cache is higher. This interference causes deterioration of the I-cache efficiency.

The above discussion of the benefits and drawbacks of inlining leads to an intuitive criteria to decide which call sites are good candidates for profitable inlining. The benefits of inlining (elimination of function call overhead, enabling of more optimization opportunities, improving cache efficiency) depend on the execution frequency of the call site. The more frequently a call site is invoked, the more promising the inlining of the site is. If the call site is invoked only a couple of hundred times in a long execution, inlining it unlikely to produce any improvement.

On the other hand, the negative effects of inlining relate to the size of the caller and the size of the callee. Larger functions tend to have worse cache behavior and higher register pressure. Inlining large callees results in more serious code bloat, and, probably, performance degradation due to additional memory spills or conflict cache misses.

Thus, we have two basic guidelines for inlining. First, the call site must be very frequent, and, second, neither the callee nor the caller should be too large. Most of the papers that address inlining take these two factors in consideration in their inlining analysis.

In this paper we describe our experience in tunning the inlining heuristics for the Open Research Compiler (ORC). The main contributions of this paper are:

- We propose *adaptive inlining* to enable aggressive inlining for small benchmarks. Usually, small benchmarks are amenable to aggressive inlining as shown in section IV. Adaptive inlining becomes conservative for large benchmarks such as GCC because the negative effects of aggressive inlining are often more pronounced in such benchmarks.
- We introduce the concept of *cycle_density* to control the code bloat and compilation time increase.
- Our detailed experimental results show the potential of inlining. We investigate the impediments to beneficial inlining and propose approaches to remove these impediments.

The rest of the paper is organized as follows: Section II describes the existent inlining analysis in ORC and its limitations. Section III describes our enhancements of the inlining analysis (adaptive inlining and *cycle_density* heuristics) and Section IV is the performance study. Section V reviews related work. Section VI quantify impediments to inlining and discusses our ongoing research.

II. Overview of ORC Inlining

In order to control the negative effects of inlining, we should inline selectively. How do we determine whether a call site is suitable for inlining? The performance effect of inlining an edge of the call graph depends on two factors: the execution frequency of the site and the size of the callee. The problem of selecting the most beneficial call sites while satisfying the code bloat constraints can be mapped to the *knapsack* problem, which has been shown to be NP-complete [11], [17]. Thus, we need heuristics to estimate the gains and the costs of each potential inlining. ORC used profiling information to calculate the *temperature* of a call site to approximate the potential benefit of inlining an edge $E_i(p,q)$ (*i.e.* a call site in function p which calls function q in the call graph).[1]

$$temperature_{E_i(p,q)} = \frac{cycle\_ratio_{E_i(p,q)}}{size\_ratio_q} \qquad (1)$$

where:

$$cycle\_ratio_{E_i(p,q)} = \frac{freq_{E_i(p,q)}}{freq_q} \times \frac{cycle\_count_q}{Total\_cycle\_count} \qquad (2)$$

$freq_{E_i(p,q)}$ is the frequency of the edge $E_i(p,q)$ and $freq_q$ is the overall execution frequency of function q in the training execution.

$Total\_cycle\_count$ is the estimated total execution time of the application:

$$Total\_cycle\_count = \sum_{k \in PUset} cycle\_count_k \qquad (3)$$

$PUset$ is the set of all program units (*i.e.* functions) in the program, $cycle\_count_q$ is the estimated number of cycles spent on function q.

$$cycle\_count_q = \sum_{i \in stmts_q} freq_i \qquad (4)$$

where $stmts_q$ is the set of all statements of function q, $freq_i$ is the frequency of execution of statement i in the training run.

Furthermore, the overall frequency of execution of the callee q is computed by:

$$freq_q = \sum_{k \in callers_q} freq_{E_i(k,q)} \qquad (5)$$

where $callers_q$ is the set of all functions that contain a call to q.

Essentially, *cycle_ratio* is the contribution of a call graph edge to the execution time of the whole application. A function's cycle count is the execution time spent in that function, including all its invocations. $(\frac{freq_{E_i(p,q)}}{freq_q} * cycle\_count_q)$ is the number of cycles contributed by the callee q invoked by the edge $E_i(p,q)$. Thus, $cycle\_ratio_{E_i(p,q)}$ is the contribution of the cycles resulting from the call site $E_i(p,q)$ to the application's total cycle count. The larger the $cycle\_ratio_{E_i(p,q)}$ is, the more important the call graph edge.

$$size\_ratio_q = \frac{size_q}{Tatal\_application\_size} \qquad (6)$$

$Total\_application\_size$ is the estimated size of the application. It is the sum of the estimated sizes of all the functions in the application. $size_q$, the estimated size of the function q, is computed by:

$$size_q = 5 * BB\_count_q + STMT\_count_q + CALL\_count_q \qquad (7)$$

where BB_count_q is the number of basic blocks in function q and reflects the complexity of the control flow in the PU, $STMT\_count_q$ is the number of statements in q, excluding non-executable statements such as labels, parameters, pragmas, and so on, and $CALL\_count_q$ is the number of call sites in q.

The $size\_ratio_q$ is the callee q's contribution to the whole application's size.

[1] Because function p may call q at different call sites, the pair (p,q) does not define an unique call site. Thus, we add the subscript i to uniquely identify the i^{th} call site from p to q.

And the $Total\_application\_size$ is given by:

$$Total\_application\_size = \sum_{k \in PUset} size_k \tag{8}$$

With careful selection of a threshold, ORC can use *temperature* to find cycle-heavy calling edges whose callee is small compared to the whole application.

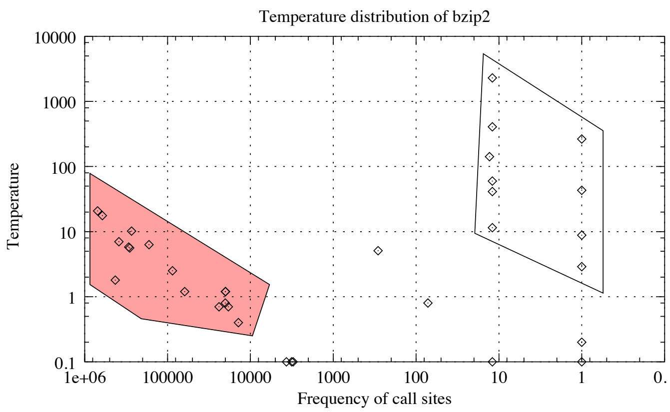

Fig. 1. Temperature Distribution of BZIP2

For instance, Figure 1 shows the distribution of the temperature for the BZIP2 benchmark.[2] The horizontal axis shows the calling frequency and the vertical axis the *temperature*. Each dot in the graph represents an edge in the call graph. The temperature varies in a wide range: from 0 to 3000. The calling frequency is shown in reverse order, the most frequently called edges appear to the left of the graph and the least frequently called are toward the right. From left to right, the temperature decreases as the frequency of the call sites also decreases. It is reasonable that the temperature doesn't go straight down because besides the call site frequency, the temperature heuristics also takes the callee's size into consideration. Procedure size negatively influences the temperature. Thus, frequently invoked call sites might be "cold" simply because they are too large.

In the original ORC inlining heuristic, an edge (call site) is rejected for inlining if its temperature is less than a specified threshold. The intuition for this heuristic is that edges with high temperature are call-sites that are invoked frequently and whose callee is small compared to the entire application. However, this heuristics may lead to the inlining of edges with high temperature but very low frequency. For instance, we highlighted two clusters of edges in the temperature×frequency graph for BZIP2 in Figure 1. The cluster to the right of the graph has higher temperature but much lower frequency than the cluster to the left of the graph. Inlining infrequently invoked call sites should always be avoided because it does not help performance. To improve this heuristic, we created a new mechanism to cooperate with the temperature heuristics to prevent inlining hot but infrequently invoked call sites. We describe our solution in Section III.

[2] To make it easy to read, the two axes of the graphs are drawn in log scale, thus some call sites whose frequencies or temperatures are 0 are not shown in the graph. The same situation exists in Figure 3.

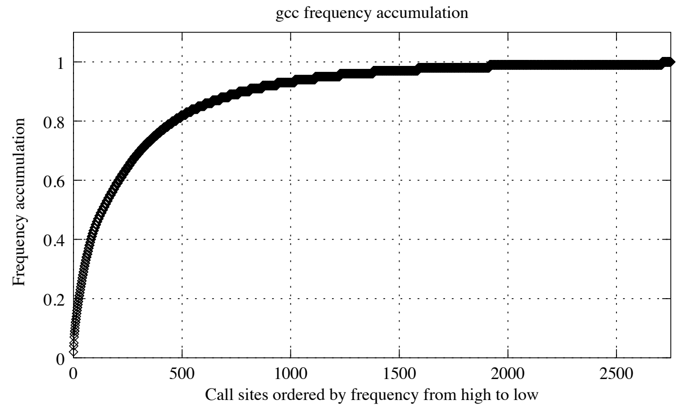

Fig. 2. Frequency accumulation of GCC (Only the top 2750 of all the 19,000 call sites are plotted.)

III. Inlining Tuning

We improve the inlining heuristics of ORC in two ways. First, adaptive inlining is employed to make the inlining heuristics more flexible. Second, a new *cycle_density* heuristics is introduced to restrict the inlining of "hot" but infrequent procedures.

A. Adaptive Inlining

The original inlining heuristic in ORC used a fixed temperature threshold (120) for inlining decisions. This threshold was chosen as a trade-off among compilation time, executable sizes and performance results of different benchmarks. However, a fixed threshold turns out to be very inflexible for applications with very different characteristics. For example, a high threshold (*e.g.* 120) is reasonable for large benchmarks because they are more vulnerable to the negative effects of code explosion resulting from inlining. However, the same threshold might not be good for small applications such as MCF, BZIP2, GZIP *etc* . We will use GCC, which is a typical large application, and BZIP2, which is a representative small application, to illustrate this problem.

Figure 2 shows the frequency accumulation for the GCC benchmark and Figure 3 shows its temperature distribution. In Figure 2, the X-axis represents the call sites sorted by invocation frequency from high to low. The i^{th} point numbered from left to right in the figure represents the accumulated percentage of the i most frequent call sites.

GCC has a very complex function call hierarchy and the function invocations are distributed amongst a large number of call sites: there are more than 19,000 call sites in GCC. In the standard SPEC2000 training execution, there are more than 42,000,000 function invocations, and the most frequent call site is called no more than 800,000 times. Figure 2 shows that the top 10% (about 2,000) most frequently invoked call sites account for more than 95% of all the function calls. Inlining these 2,000 call sites would result in unbearable compilation cost and substantial code bloat.

In Figure 3, according to the frequency of execution, we should inline the call sites on the left hand side of the graph and we should avoid inlining the call sites on the right hand side. Notice that several call sites on the right hand side are hot, and thus are inlined by the original heuristics of ORC.

For large applications, the improvement from inlining is usually very limited (as we will see in the section IV). On one hand, it is impossible to eliminate most of the function overheads without wholesale inlining. On the other hand, if we use the same temperature threshold as for small benchmarks, we might end up with the problem of *over-inlining*, *i.e.* too many procedures are inlined and the negative effects of inlining are more pronounced than the positive ones. For example, if the temperature threshold is set to 1, there will be more than 1,700 call sites inlined in GCC. Such aggressive inlining makes the compilation time much longer without performance improvement as our experiments show.

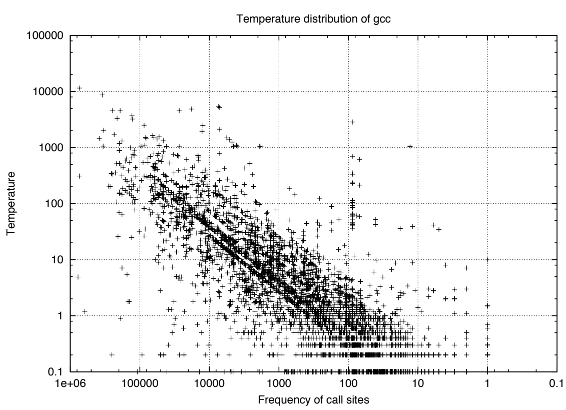

Fig. 3. Temperature Distribution of GCC

The high temperature threshold (120) in the original ORC was chosen to avoid over-inlining in large applications. However, this conservative strategy impedes aggressive inlining for small benchmarks where code bloat is not as prominent. For instance, Figure 1 and Figure 4 show the temperature distribution and frequency accumulation of the BZIP2 benchmark. There are only 239 call sites and about 3,900 lines of C code in BZIP2. This implies that the program is quite small (compared to more than 19,000 call sites and 190,000 lines of C code in the GCC benchmark). Moreover, in BZIP2 the top ten most frequently invoked call sites (about 4.2% of the total number of call sites) accounts for nearly 97% of all the function calls (Figure 4).

As we will see in the section IV, aggressive inlining is good for small benchmarks such as BZIP2: inlining the 10 most frequently invoked call sites in BZIP2 eliminates almost all the function calls.

However, the inflexible temperature threshold often prevents the inlining of the most frequent call sites (the points in the shadowed area in Figure 1) because their temperatures are lower than the fixed threshold (120). Thus, it is desirable that the temperature threshold for small benchmarks be lowered because many of the call sites that have performance potential do not reach the conservative temperature threshold used to prevent code bloat in large applications.

The contradiction between the threshold distributions of large benchmarks and small ones naturally motivates adaptive inlining: we use high temperature threshold for large applications because they tend to have many "hot" call sites; and we enable more aggressive inlining for small applications by lowering the temperature threshold for them.

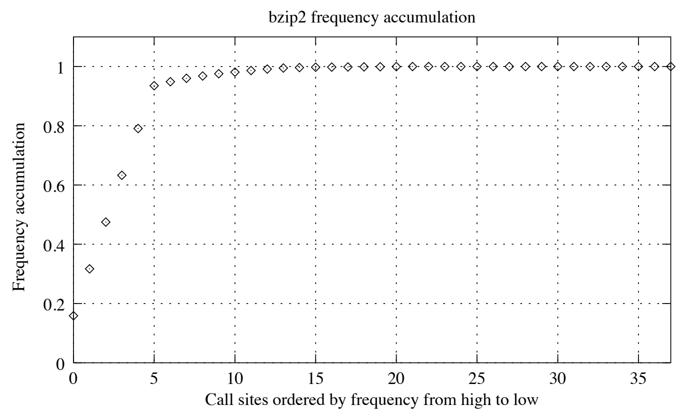

Fig. 4. Frequency accumulation of BZIP2 (Only the top 38 of all the 239 call sites are plotted.)

Adapting the inlining temperature threshold according to application size is pretty simple in ORC. Because the estimated size of each procedure in ORC is available in the Inter-Procedural Optimization (IPO) phase, their sum is the estimated size of the application.[3] We classify applications into three categories: large applications, median applications and small applications. In the compilation, we utilize proper temperature threshold according to the estimated application size. If an application is a large application, its temperature threshold is 120. If it is a median application, its temperature threshold is 50. Otherwise, the temperature threshold is lowered to 1. The threshold values were obtained by a detailed empirical study of the SPEC2000 benchmarks.[4] This division of applications into three categories produces better results than any single threshold applied to all benchmarks.

B. *Cycle_density*

The intuition behind the definition of temperature is that hot procedures should be frequently invoked and not too large. However, as we have seen in Figure 3 and Figure 1, some of the procedures with high temperature are

[3] We ignore library functions and dynamic shared-objects because we cannot acquire this information at compilation time.

[4] This approach is not unlike the application of machine learning to tune compilers used in [18]. However in our case we chose the parameter through manual tuning.

```
// decide if a call site should be inlined (returning TRUE)
// or not (returning FALSE).
BOOL inlining_analysis(call_site)
{
    // MEDIAN_THRESHOLD & LARGE_THRESHOLD are pre-selected thresholds
    // to classify the application as large, small or median,
    // and select the proper temperature threshold

    if ( estimated_size < MEDIAN_THRESHOLD)
        temperature_threshold = 1;
    else
    if ( estimated_size < LARGE_THRESHOLD)
        temperature_threshold = 50;
    else
        temperature_threshold = 120;

    // temperature_analysis() computes the temperature of a call site
    // and compare it with the temperature threshold. It returns TRUE
    // if this call site is "hot" enough for inlining and FALSE otherwise.
    if (temperature_analysis(temperature_threshold, call_site)) {

        // if this is the only call to the callee
        // in the entire application, ORC inlines it anyway
        if (called_only_once(callee))
            return TRUE;

        // cycle_density_analysis() computes the cycle_density of the callee
        // and compare it with the cycle_density threshold to decide whether
        // the callee is ``heavy'' or not
        if (cycle_density_analysis(call_site)) {
            return TRUE;
        }else {
            return FALSE;
        }
    }else {
        // do not inline this call site
        return FALSE;
    }
}
```

Fig. 5. Adaptive inlining in ORC.

not actually "hot", *i.e.* some infrequently invoked call sites also have high temperatures (those points in the top-right part of the graphs). These call sites correspond to functions that are not called frequently, but contain high-trip count loops that contribute to their high *cycle_ratio*, which result in a high temperature (see Equation 2). We call the functions that are called infrequently but have high temperatures *heavy functions.*

Inlining heavy functions results in little performance improvement. First, very few runtime function calls are eliminated. Second, the path from the caller to a heavy function is not a hot path at all, and thus will not benefit from post-inlining optimization. Third, inlining heavy functions might prevent frequent edges from being inlined if the code growth budget is spent. To handle this problem, we introduce *cycle_density* to filter out heavy functions.

$$cycle\_density_q = \frac{cycle\_count_q}{frequency_q} \tag{9}$$

where $cycle\_count_q$ is the number of cycles spent on procedure q and $frequency_q$ is the number of times that the procedure q is invoked.

When a call site fulfills the temperature threshold, the *cycle_density* of the callee is computed. If the callee has a large cycle count but small frequency, *i.e.* its *cycle_density* is high, it must contain loops with high trip count. These heavy procedures are not inlined. *cycle_density* has little impact on the performance because it only filters out infrequent call sites. However, *cycle_density* can significantly reduce the compilation time and executable sizes, which is important in some application contexts, such as embedded computing.

Figure 6 compares the temperature against the *cycle_density* for each call site in BZIP2. For call sites that are actually "hot", the temperature is indeed high while the *cycle_density* is low (for BZIP2 they are always less than 0.5). These call sites are the ones that will benefit from inlining.

Infrequently invoked call sites fall into two categories according to their temperatures. Infrequently invoked call sites with low temperature are eliminated by the temperature threshold. Infrequently invoked call sites with high temperature always have very high *cycle_density*. Thus we can prevent the inlining of these sites by choosing a proper *cycle_density* threshold. In our tuning, we use a fixed *cycle_density* threshold of 10 that works well for the SPEC2000 benchmarks as we will see in the next section.

We implemented this enhanced inlining decision criteria

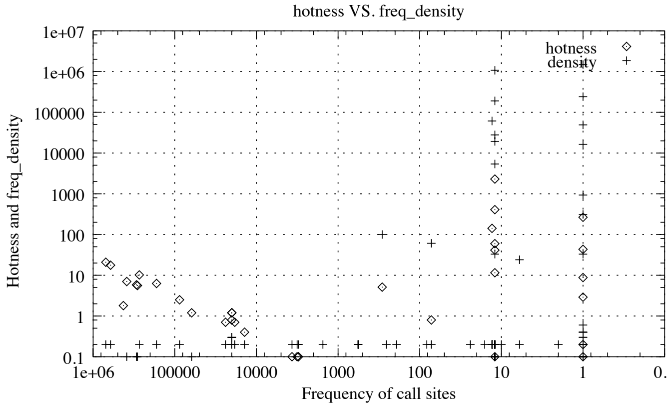

Fig. 6. Cycle Density VS. Temperature (BZIP2)

and contributed it to the ORC-2.0 release. Figure 5 shows the C-style pseudo code for the improved inlining analysis in the ORC. Notice that a procedure that has a single call site in the entire application will always be inlined. The reasoning is that the inlining of that single call site will render the callee dead, and will allow the elimination of the callee, therefore this inlining will save function invocations without causing code growth.

IV. Results

A. Experimental Environment

We investigate the effects of adaptive inlining and of the introduction of the *cycle_density* heuristic on performance, compilation time, and the final executable size of SPEC INT2000 benchmarks. We use a cross-compilation method: we run ORC on an IA32 machine (a SMP machine with 2 Pentium-III 600MHz processors and 512MB memory) to generate an IA64 executable which is run on an Itanium machine (733MHz Itanium-I processor, 1GB memory). Thus our performance comparison is conducted on the IA64 systems and our compilation time comparison is conducted on the IA32 system.

B. Performance Analysis

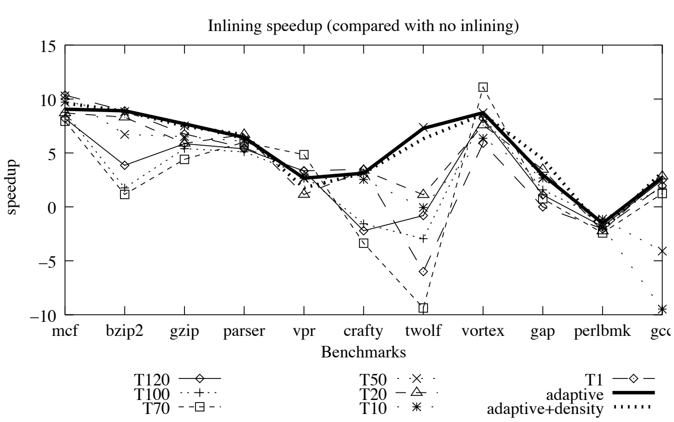

Fig. 7. Overall performance comparison

Figure 7 shows the performance improvement when different inlining strategies are used. `T120` represents a fixed temperature threshold of 120, `T1`, is a fixed temperature threshold of 1, similarly for the other `T` labels. In `adaptive` the temperature threshold varies according to the *adaptation* heuristic described in Section II. In the `adaptive+density` compiler, both the adaptation and the *cycle_density* heuristics are used.

Except for PERLBMK, in all benchmarks the adaptation heuristic results in positive speedup for inlining.[5] These results suggest that our adaptive temperature threshold is properly selected. In some cases the difference between a fixed threshold and the threshold chosen with adaptation is very significant (see BZIP2 and TWOLF). Note also that the addition of *cycle_density* to adaptation does not produce much effect on performance. This result is explained by the fact that *cycle_density* only prevents heavy and infrequently invoked functions from inlining.

We arranged the benchmarks in Figure 7 according to their sizes with the smaller benchmarks on the left and the larger ones on the right. Comparatively, in general, for small benchmarks inlining yields better speedups than for large benchmarks. This observation can be made by examining the maximum performance improvement from all the strategies. Excluding TWOLF and VORTEX, the maximum performance improvement decreases from left to right (from small benchmark to large benchmarks). This trend suggests a loose correlation between the application size and potential performance improvements that can be obtained from inlining.

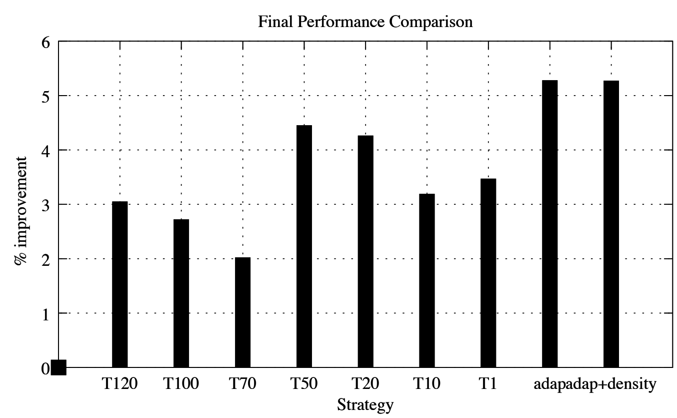

Fig. 8. Final Performance Comparison

Figure 8 compares the performance improvements of different strategies more explicitly. Each bar represents the average performance speedup for the 11 benchmarks studied. The base line is the average performance of the 11 benchmarks compiled without inlining. And the two rightmost bars are for adaptive inlining without and with *cycle_density* heuristics. Adaptive inlining strategy speeds up the benchmarks by 5.28%, while the best average per-

[5] Inlining seems to always have a slight negative effect on the performance of PERLBMK. We are currently investigating this benchmark in more detail.

	Executable Size					Compilation Time		
Benchmarks	no inline (Bytes)	adaptive		adap+density		no inline (Secs)	adaptive	adap+density
		% inc	calls	% inc	calls		% increase	% increase
bzip2	116295	54.1	89	26.9	88	70.356	117.8	71.3
gcc	4397983	4.4	919	4.4	919	4194.54	6.0	4.0
crafty	635855	20.1	204	20.1	204	440.687	30.9	30.9
gap	1977644	9.7	345	7.3	343	1409.18	9.1	2.7
gzip	147417	67.6	62	28.0	54	109.457	93.8	41.2
mcf	48241	-0.5	19	-6.3	17	41.832	9.3	8.5
parser	340223	18.1	239	16.4	224	274.868	17.1	12.9
perlbmk	2163047	7.5	419	7.5	419	1518.37	10.6	8.9
twolf	823832	10.6	147	10.6	147	646.769	19.8	20.5
vortex	1170014	31.4	210	31.1	208	1162.27	33.0	36.5
vpr	532912	17.5	141	16.4	139	293.683	30.2	26.2
average		21.9		14.8			34.3	24.0

TABLE I

cycle_density'S IMPACT ON EXECUTABLE SIZE AND COMPILATION TIME

formance gain of all other strategies is 4.45% when the temperature threshold is 50. Notice also that the performance influence of *cycle_density* heuristics is negligible.

C. Compilation Time and Executable Size Analysis

In this section, we study the effect of the *cycle_density* heuristics on the compilation time and on the executable size. Because *cycle_density* filters procedures that have high temperatures but are infrequently invoked call sites, we expected that its use should reduce both the compilation time and the final executable size.

Table I shows the executable size, measured in bytes, and the compilation time, measured in seconds, for all benchmarks when no inlining is performed. Then for the compiler with adaptive inlining and the compiler with adaptive inlining with *cycle_density*, the table displays the percentage increase in the executable size and on the compilation time. The table also show, under the "calls" columns, the number of call sites that were inlined in each case.

The *cycle_density* heuristic significantly reduces the code bloat and compilation time problem. On average, adaptive inlining increases the code size by 21.9% and the compilation time by 34.3%. When *cycle_density* is used to screen out heavy procedures, these numbers reduce to 14.8% and 24%, respectively. It is also interesting to compare the actual number of inlined call sites: the *cycle_density* heuristic only eliminates a few call sites. Except for GZIP and PARSER, *cycle_density* prevents the inlining of no more than 2 call sites in each benchmark. Table I also shows some curious results. Although *cycle_density* prevents the inlining of a single call site for BZIP2, the code growth reduces from 54.1% to 26.9%. A close examination of BZIP2 reveals that the procedure *doReversibleTransformation* calls *sortIt* infrequently (only 22 times in the standard training run). However ORC performs a bottom-up inlining, in which the edges in the bottom of the call graph are analyzed and inlined first. In the BZIP2 case, *sortIt* absorbs many functions and becomes very large and *heavy* before it is analyzed as the callee. When ORC analyzes the call sites that have *sortIt* as the callee, the estimated cycle number spent in *sortIt* is huge, which contributes to its high temperature. However, *sortIt* is called infrequently and its inlining does not produce measurable performance benefits. *cycle_density* filters these heavy functions successfully.

Finally, *cycle_density* only eliminates a few call sites because it is not applied to callees that are only called at one call site in the entire application (see Figure 5).

V. Related Work

In this paper we presented improvements to the inlining heuristics in the Open Research Compiler (ORC). Several researchers have investigated inlining. However, very few of them produced a detailed empirical study on an industry-strong compiler infrastructure based on industry-standard benchmarks such as the one that we present in this paper.

Ayers *et al.* [2] and Chang *et al.* [5], [13] demonstrate impressive performance improvement by aggressive inlining and cloning. Their inlining facility is very much like that in ORC: the inlining happens on high level intermediate representation, using feedback information and cross-module analysis. Both of them use a budget to control code bloat: inlining a call site consumes code growth budget. Ayers *et al.* use an estimated 100% compilation time increase as budget for inlining. ORC uses an estimated 100% code size increase for the inlining budget. In our experiments, inlining in ORC never uses up the budget (i.e. double the estimated application size).

Without feedback information, Allen and Johnson perform inlining at source level [1]. Besides reporting impressive speedup (12% in average), they also show that inlining might exert negative impact on performance.

Several researchers try to enable aggressive inlining in the context of object oriented programming. A single call site may have multiple potential callees. For instance, C and C++ programming languages allow calling functions through pointers. Polymorphism in OO programming languages is often realized via indirect function calls, also called virtual method invocation. For indirect function calls, it may be impossible to infer the callee before runtime. Thus, inlining cannot be applied straightforwardly to dynamic function calls. A series of special inlining approaches were developed to improve the performance of applications that employ indirect function calls intensively [3], [4], [9], [10], [12].

VI. Ongoing Work

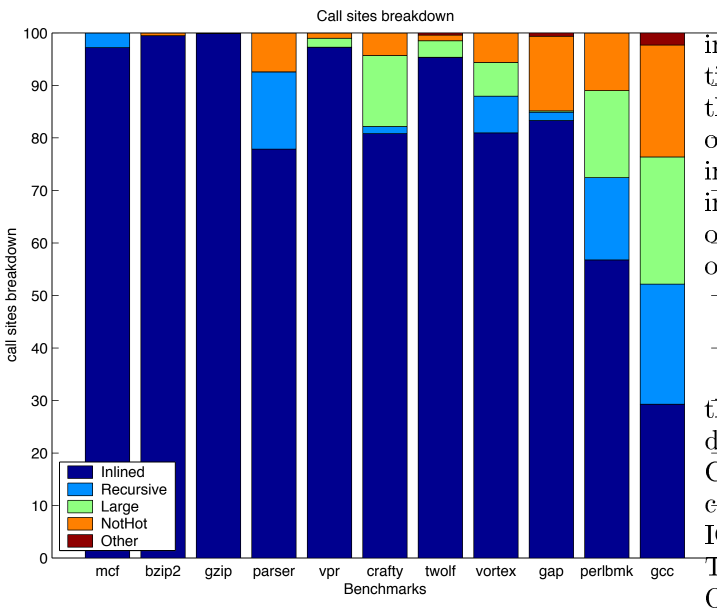

Fig. 9. Call Site Breakdown

Figure 9 shows how many dynamic function calls we can eliminate using our adaptive inlining technique. We divided the function calls into five different categories:

inlined Call sites that can be inlined with our adaptive inlining technique. These call sites have high temperature and low *cycle_density*.

NotHot Call sites that are not frequently invoked. It brings no benefit to inline these call sites.

Recursive ORC does not inline call sites that are in a cycle in the call graph.

Large Call sites that have high temperature but cannot be inlined because either the callee, the caller or its combination is too large. GCC, PERLBMK, CRAFTY and GAP have some large call sites.

Other Call sites that cannot be inlined due to some other special reasons. For example, the actual parameters to the call sites do not match the formal parameters of the callee. As Figure 9 shows, these call sites are very rare.

With our enhanced inlining framework, we were able to eliminate most of the dynamic function calls for small benchmarks such as MCF, BZIP2 and GZIP. However we only eliminated about 30% dynamic function invocations for GCC and 57% for PERLBMK. Examining the graph in Figure 9, to obtain further benefits from inlining we need to address inlining in these large benchmarks. The categories that are the most promising are the recursive function calls and call sites with large callers or callees.

Figure 9 shows that for some large benchmarks (PARSER, PERLBMK and GCC) a significant portion of the function invocations that are not inlined are recursive functions. We are conducting a study of the depth of recursions. If a recursive function is invoked often, but its recursion is shallow, limited inlining should be beneficial (the analogy in intra-procedural transformations is loop unrolling).

In order to harvest the benefits from inlining without incurring in high costs on code growth and compilation time, we should inline only the portions of the procedures that are actually hot. Thus we are currently investigating outlining-enabled inlining (*i.e.* partial inlining). Partial inlining has been proposed in [15], [16]. Our initial studies indicate that there is potential performance gains to be obtained from partial inlining, and thus this is the focus of our ongoing research.

VII. Acknowledgements

We had a lot of help to perform this work. Most of the heuristics analysis and performance tunning were done during Peng Zhao's internship in the Intel China Research Center (ICRC). We thank the SGI team for making the code that originated ORC open source. We thank the ICRC and the ORC team in the Institute of Computing Technology, Chinese Academy of Sciences for building the ORC research infrastructure. Intel generously donated us the Itanium machine used in the experiments. Sincere thanks to Sun C. Chan, Roy Ju and Guei-Yuan Lueh for their help and discussion on the ORC inlining tuning. Thanks also go to people who answered a lot of questions in the ORC and Open64 mailing lists. This research is supported by the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 241–249, 1988.

[2] Andrew Ayers, Robert Gottlieb, and Richard Schooler. Aggressive inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1997.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 324–341, 1996.

[4] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 397–408, Portland, Oregon, 1994.

[5] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Software - Practice and Experience*, 22(5):349–369, 1992.

[6] J. W. Davidson and A. M. Holler. A model of subprogram inlining. Technical report, Computer Science Technical Report TR-89-04, Department of Computer Science, University of Virginia, July 1989.

[7] Jack W. Davidson and Anne M. Holler. A study of a C function inliner. *Software - Practice and Experience (SPE)*, 18(8):775–790, 1989.

[8] Jack W. Davidson and Anne M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering (TSE)*, 18(2):89–102, 1992.

[9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–101, Arhus, Denmark, August 1995.

[10] David Detlefs and Ole Agesen. Inlining of virtual methods. In *13th European Conference on Object-Oriented Programming (ECOOP)*, June 1999.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[12] K. Hazelwood and D. Grove. Adaptive online context-sensitive inlining. In *Internetaional Symposium on Code Generation and Optimization*, pages 253–264, San Francisco, CA, March 2003.

[13] W. W. Hwu and P. P. Chang. Inline function expansion for compiling realistic c programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1989.

[14] Rainer Leupers and Peter Marwedel. Function inlining under code size constraints for embedded processors. In *International Conference on Computer-Aided Design (ICCAD)*, Nov 1999.

[15] Robert Muth and Saumra Debray. Partial inlining. Technical report, Dept. of Computer Science, Univ. of Arizona, U.S.A., 1997.

[16] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1990.

[17] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, Jan 1977.

[18] M. Stephenson, S. Amarasinghe, M. Martin, and U. O'Reilly. Meta-optimization: Improving compiler heuristics with machine learning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.

[19] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. An empirical study of method inlining for a Java just-in-time compiler. In *2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, Aug 2002.