SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:909–933
Published online 23 November 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.792

Toward a progress indicator for program compilation

Gang Luo*,†, Tong Chen and Hao Yu

IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, U.S.A.

SUMMARY

For user-friendliness purposes, many modern software systems provide progress indicators for long-running tasks. These progress indicators continuously estimate the percentage of the task that has been completed and when the task will finish. However, none of the existing program compilation tools provide a non-trivial progress indicator, although it often takes minutes or hours to build a large program. In this paper, we investigate the problem of supporting such progress indicators. We first discuss the goals and challenges inherent in this problem. Then we present a set of techniques that are sufficient for implementing a simple yet useful progress indicator for program compilation. Finally, we report on an initial implementation of these techniques in GNU Make. Copyright © 2006 John Wiley & Sons, Ltd.

Received 6 March 2006; Revised 22 August 2006; Accepted 29 August 2006

KEY WORDS: progress indicator; program compilation; tool

1. INTRODUCTION

Progress indicators (PIs) are a widely used user-interface technique in modern software systems. For example, Figure 1 shows a PI for file downloading.

A typical PI has the following two features. First, it continuously estimates how much of the task has been completed. Second, it continuously estimates the remaining task execution time. With these two features, users can have an idea of how long they need to wait for the task to finish and better utilize their time [1]. Hence, the software systems become more user-friendly. In fact, according to some Human Computer Interaction (HCI) authorities, every task that runs for longer than 10 s needs a PI [2]. Moreover, in a large number of cases, users can benefit from even a rough estimate of the remaining task execution time [3]. Consequently, PIs are being used in more and more software systems. For example, due to users' strong requests, Microsoft and IBM are currently incorporating sophisticated PIs into SQL Server [4] and DB2 [5] relational database management systems (RDBMSs), respectively.

*Correspondence to: Gang Luo, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, U.S.A.
†E-mail: luog@us.ibm.com

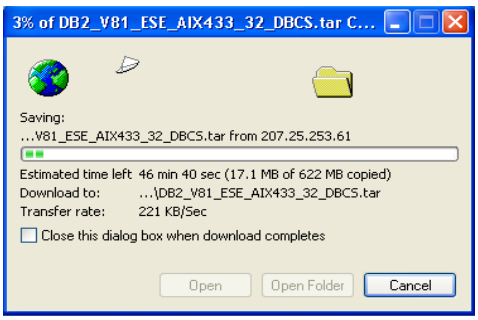

Figure 1. A typical file download interface.

PIs are also desirable in program compilation. Although computers are becoming faster and faster, it still often takes a long time to build a large program. This is because programs are becoming increasingly larger, and compilers need to perform more transformations and optimizations due to the increasing use of higher-level programming model (e.g. classes, templates, and middleware). Also, to achieve better performance and generate more efficient code for the new-generation, multi-core CPUs, compilers are becoming more aggressive in applying new optimizations that were not used previously.

For many people, building large programs has become a daily activity these days. In industry, a common scenario is that a group of software engineers collaborate on a large software project. Every once a while (e.g. to fix a bug) for safety purposes, a software engineer needs to obtain the entire latest version of the software and build the package from scratch. This often takes minutes or even hours. Such a scenario also occurs frequently in the open source software community. For example, a group of Gentoo Linux users [6] list the build time of several open source software programs. In that list, the build time of the Open Office software on a 2.4 GHz Xeon computer is around 11 h. As a second example, due to the heavy burden of building programs, Oracle (the world's second-largest software company) has a Grid system with more than 1000 computers dedicated to program building (and some to testing) [7].

In practice, even if a large program has been built before, once some source code files get modified, we need to first re-compile all the modified source code files into object files, and then re-link all the object files to generate a new executable file. In particular, in the case that a header file gets modified, all source code files that are related to this header file need to be re-compiled. Re-compiling a single source code file may take several seconds and re-compiling multiple source code files may take several minutes, which is quite long from users' perspective. For example, the Open Office software contains less than 8000 source code files. According to the build time reported in [6], on a 2.4 GHz Xeon computer, the average compilation time of a source code file is around 5 s (11 h per 8000 files).

Besides providing a user-friendly interface, PIs have other potential uses.

(1) *Load management*. Suppose that multiple programs are being built on a single system (e.g. Oracle uses a Grid system to build all programs submitted by the developers [7]) and that for some reason the execution of some program build tasks needs to be blocked so that

DOI: 10.1002/spe

other programs can be built faster. A PI can help us decide which program building tasks to be blocked.

(2) *Automatic administration*. In certain cases, a program build task needs to be finished before a deadline. A PI can help us choose an appropriate level of optimization to ensure a reasonable program build time.

Unfortunately, although PIs are desirable in program compilation, to the best of our knowledge, none of the existing program compilation tools provides a non-trivial PI, and we are not aware of any published techniques for supporting such a PI.

Due to the lack of system support, users have already started to build their own PIs [8–11]. These PIs are trivial in the sense that they only count the number of source code files compiled so far. Then they report the percentage of the program build task that has been completed as the ratio of the number of compiled source code files to the total number of source code files that need to be compiled. While such a PI is clearly much better than nothing, for many purposes it will be too coarse. First, the compile time of different source code files may be very different. (In this paper, we refer to the sum of the compile time and the link time as the program build time.) Second, users are interested in the remaining program build time. However, it is not reported. Finally, the link time is not considered.

Another way to provide a trivial PI is to let the program compilation tool estimate the program build time (cost) before we start building the program [12]. Providing a trivial PI based upon this estimate is simple. If the program compilation tool estimates that building a program will take t seconds, and the program build task has run for t' seconds, the remaining time is estimated to be $t - t'$ seconds. While such a trivial PI is also better than nothing, it is likely to be highly inaccurate. This inaccuracy arises from two main causes. First, the program build cost estimated by the program compilation tool is typically not precise [12]. Second, due to other concurrently running jobs, the system load may vary significantly. For a specific program build task, even if the program compilation tool provides an estimate that is precise for an unloaded system, this estimate may differ substantially from the actual program build time in a loaded system.

In this paper, we propose techniques for supporting PIs for program compilation. The utility of these techniques is demonstrated by a prototype implementation in GNU Make, a widely used open-source program compilation tool [13,14]. While the resulting PI can be refined, our experiments show that it is a useful PI even in the presence of program compilation tool estimation errors and varying run-time system loads, and that it imposes a negligible penalty on the running time of program build tasks.

Our basic approach is to begin with the program build cost estimated by the program compilation tool. However, as a program is being built, we obtain more precise information about the program build task and continuously refine the estimated program build cost. Also, at all times, we monitor the program build speed (i.e. the execution speed of the program build task), which is a function of the program and the system load. This more precise information is used to continuously refine the estimated program build time and thus to update the PI.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the goals of PIs for program compilation. Section 4 gives a set of techniques for implementing PIs in a program compilation tool. Section 5 presents results from an initial implementation of our techniques in GNU Make. Our conclusions are given in Section 6.

DOI: 10.1002/spe

2. RELATED WORK

There has been a lot of work (e.g. Myers [1] and Berque and Goldberg [3]) in the HCI community for PIs. However, none of this work has addressed program compilation.

Braun [15] proposed dynamically controlling the program build time. That work is contrary to our work of continuously refining the estimated program build time.

Flajolet and Steyaert [16], Hickey and Cohen [17], Sarkar [18], and Wegbriet [19] proposed several techniques for estimating the program execution time. Compared to general program execution, program building has its own characteristics and hence more precise cost models can be built [12]. This will increase the precision of the PI's estimates.

Supporting PIs for SQL queries in RDBMSs has been proposed in Chaudhuri *et al*. [4] and Luo *et al*. [20]. Executing SQL queries in a RDBMS and building programs have different performance characteristics. For example, the cost formula of a SQL query [21] is different from that of building a program. Hence, the techniques proposed in Chaudhuri *et al*. [4] and Luo *et al*. [20] cannot be used directly for program compilation.

Ilyas *et al*. [22] proposed a method for estimating the optimizer's compilation time of a SQL query in a RDBMS. There, SQL queries are optimized using a dynamic programming technique [23], which is different from the techniques used for building programs. As a result, the method proposed in Ilyas *et al*. [22] does not apply to program compilation.

3. GOALS FOR PROGRESS INDICATORS

Figure 2 shows an example of the sort of PI we would like to support for program compilation. This interface, which is continuously updated, displays the elapsed time, the estimated remaining program build time, the estimated percentage of the program build task that has been completed, the number and the percentage of source code files that have been compiled, the estimated program build cost, and the current program build speed. The estimated program build cost is measured in terms of U, where U is an abstract quantity that represents one unit of work (we will return to the question of how to define U in Section 4). The current program build speed is also measured by U. This example is for the windowing environment. In a command line environment, all the texts in the PI are still displayed but the graphics no longer exists.

Ideally, a PI should satisfy the following four goals [20].

- *Continuously revised estimates*. At any time, for all the information provided to the user, the PI should give an estimate based on all the information available about the program build task and the system at that time. This estimate should be continuously refined, through the use of more precise information about the program build task and changes in the rate at which the program build task is progressing.
- *Acceptable pacing*. The PI should be updated frequently enough so that the user sees a smooth display. However, the update rate should not be so frequent as to overburden either the user interface or the user.
- *Minimal overhead*. The PI should have a small effect on the efficiency of the program build task.
- *Maximal functionality*. A typical program compilation tool has a large number of combinations of options. The PI should provide useful information for every combination of options.

DOI: 10.1002/spe

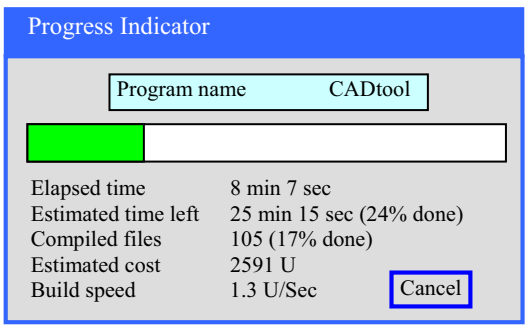

Figure 2. A PI for program compilation.

4. MODELING COST METHOD

In this section, we present our techniques for implementing PIs in a program compilation tool. Our main idea is as follows. First, we collect profiles. As a program is being built, we will have more precise information about the program build task. This improved information is used to continuously refine the estimated program build cost. Also, the program build speed (i.e. how many U are processed per second) is continuously monitored. At any time, the remaining program build time is estimated to be the ratio of the estimated remaining program build cost to the observed current program build speed. From time to time, the PI presents the latest estimates to the user. As our method is based on sophisticated cost models, we call it the *modeling cost method*.

Section 4.1 describes how to choose the work unit U and how it is converted to time. Section 4.2 shows how to continuously refine the estimated program build cost. Section 4.3 discusses the techniques used in monitoring the program build speed. Section 4.4 presents the techniques for handling different combinations of optimization flags.

4.1. Choosing U and converting to time

As mentioned in Section 3, the estimated program build cost is measured in the abstract unit U, where each U represents one unit of work. The current program build speed is also measured by U. We are purposely being rather vague and general in this statement, as many viable alternatives exist for U. The important requirements for U are that one can readily estimate how many U a program build task will take to execute, and that one can readily convert from U to estimated time, since ultimately time is likely to be the unit most meaningful to users. Reasonable candidates for U include CPU cycles, I/Os, or even a combination of the two, perhaps using some weighting factor.

Our PI works by continuously refining both its estimate of the program build cost and its estimate of the conversion factor from U to time. As the program is being built, more profiles are gathered about the program and the estimated number of U required to build the program changes.

DOI: 10.1002/spe

The refinements in the estimates of the conversion factor from U to time result from observations of how quickly the system is processing U.

In this paper, for simplicity, U is set to be 1×10^9 CPU cycles. Initially (before we start building the program), we assume that building the program will require a number of U equal to the program compilation tool's initial estimate. Before giving its first estimate of running time, the PI 'watches' some amount of processing to see how quickly the system is consuming U (this is discussed in more detail in Section 4.3). As the program is being built, the estimated time to process one U will change to reflect the observed processing rate in the system. In Section 5, we show that this simple definition of U works well.

4.2. Refining the estimated program build cost

In this section, we describe techniques for refining the estimate of the number of U the program build task will require. We assume that the program contains n ($n \geq 1$) source code files (header files are not included). The size of the ith ($1 \leq i \leq n$) source code file is s_i. We discuss the case that the entire program is built from scratch. The case that some of the n source code files get modified and need to be re-compiled can be discussed in a similar way (the differences are pointed out in the last second paragraph before Section 4.3).

4.2.1. Cost model

The program build task performs the following two steps:

(1) compile the n source code files into n object files;
(2) link the n object files into an executable file.

Let c_{build} denote the cost of the program build task. The cost of step (1) is c_{compile}. The cost of step (2) is c_{link}. The cost of compiling the ith ($1 \leq i \leq n$) source code file into an object file is c_i. Then

$$c_{\text{build}} = c_{\text{compile}} + c_{\text{link}} \quad \text{and} \quad c_{\text{compile}} = \sum_{i=1}^{n} c_i$$

Using Halstead's software complexity model, Shaw *et al*. [12] derived a heuristic formula for c_{compile}:

$$c_{\text{compile}} = K \times V^a$$

where V is roughly equal to the number of tokens in the program, and K and a are two factors. In our case, as mentioned in Section 3, we want the PIs to have minimal overhead. Hence, it is not desirable to parse the program to obtain the number of tokens. Rather, file sizes (s_i) are used to approximate the number of tokens. More specifically, the following heuristic formula is used to estimate c_i ($1 \leq i \leq n$):

$$c_i = K_c \times s_i^{L_c}$$

where K_c and L_c are two factors (the subscript c stands for compile). Similarly, the following heuristic formula is used to estimate c_{link}:

$$c_{\text{link}} = K_l \times S^{L_l}$$

where K_l and L_l are two factors (the subscript l stands for link), and $S = \sum_{i=1}^{n} s_i$. How to set the values of K_c, L_c, K_l, and L_l will be discussed below. Later in Section 5.2, we show that these formulas model reality well.

DOI: 10.1002/spe

4.2.2. *Setting initial values*

The developer of the program compilation tool provides k ($k \geq 1$) training programs. We first assume that all programs (including both these k training programs and users' programs) are built using the same optimization flags. Later in Section 4.4, we describe how to handle different combinations of optimization flags.

The program compilation tool computes the initial values of the four factors K_c, L_c, K_l, andL_l based on its experience with these k training programs. When the program compilation tool builds these k training programs, it measures the value of c_i and c_{link}. Then the program compilation tool uses linear regression [24] to compute the initial values of K_c, L_c, K_l, and L_l:

$$\ln c_i = \ln K_c + L_c \times \ln s_i \quad \text{and} \quad \ln c_{\text{link}} = \ln K_l + L_l \times \ln S$$

As described below, when we start to use the program compilation tool to build programs, the values of the four factors K_c, L_c, K_l, and L_l are continuously adjusted, as the program to be built may have different characteristics from the k training programs. After building the program one or more times, the values of the four factors K_c, L_c, K_l, and L_l will fit well with the program.

At this point, the values of the four factors K_c, L_c, K_l, and L_l still need to be continuously adjusted. This is because the user may switch to building a new program, and the values of the four factors K_c, L_c, K_l, and L_l need to continuously evolve to fit with the new program.

4.2.3. *Refining procedure*

As mentioned in Section 3, we want to update the display on PIs as smoothly as possible. Hence, the estimates of c_i ($1 \leq i \leq n$) and c_{link} need to be continuously refined. We only do this refinement each time we finish compiling a source code file rather than during the compilation of a source code file (note that the estimated remaining program build time is continuously being updated). This is to reduce the overhead of PIs. Since the compilation time of a single source code file (say, 5 s) is typically small compared to the entire program build time (say, several minutes or hours) this is also acceptable.

Suppose so far, in step (1) (the compiling step), the first m ($m \leq n$) source code files have been compiled into object files. Then the exact c_i ($1 \leq i \leq m$) is known. Therefore, we only need to focus on c_j ($m + 1 \leq j \leq n$).

Linear regression is used to compute K_{c1} and L_{c1}:

$$\ln c_i = \ln K_{c1} + L_{c1} \times \ln s_i \quad 1 \leq i \leq m$$

That is, K_{c1} and L_{c1} are computed based on the first m source code files that have been compiled into object files. Let the percentage of all source code files that have been compiled be given by $p = m/n$. The following heuristic formula is used to estimate c_j ($m + 1 \leq j \leq n$):

$$c_j = K_{c2} \times s_j^{L_{c2}}$$

where $K_{c2} = p \times K_{c1} + (1 - p) \times K_c$, and $L_{c2} = p \times L_{c1} + (1 - p) \times L_c$.

This heuristic formula intends to smooth fluctuations in the estimator and let it gradually change from the initial estimate (which is computed based on K_c and L_c when $p = 0$) to the estimate that is computed based on the current program's compilation (when $p = 1$).

DOI: 10.1002/spe

After step (1) (the compiling step) is finished, K_c and L_c are replaced with K_{c2} and L_{c2}, respectively. The new values of K_c and L_c will be used for the next compilation. Note: if we build the entire program, then after step (1) is finished, we have $p = 1$. However, if only n_1 ($n_1 < n$) of the n source code files get modified and re-compiled, then after step (1) is finished, we have $p = n_1/n < 1$.

Now we consider step (2) (the linking step). Throughout the entire execution of step (2), c_{link} is estimated as

$$c_{\text{link}} = K_l \times S^{L_l}$$

After step (2) is finished, linear regression is used to re-compute K_l and L_l based on the compilation of the last k programs. (At the beginning, the k training programs are treated as the last k programs.) The new values of K_l and L_l will be used for the next program's compilation.

4.3. Monitoring current and predicting future program build speed

Recall that our PI depends on two things: the estimates of U, and the estimated conversion factor between U and time. The conversion of U to time should reflect what we are observing as the system is running. So, at all times, we keep track of the amount of work (measured in U) that has been done for the program build task in the last T seconds, where T is a pre-defined number. This is achieved by performing system calls to obtain the profiles of the compiling/linking processes that are collected by the operating system. The average program build speed in the last T seconds is used as the estimated current program build speed. To minimize the influence of temporary fluctuations, this T should not be too small. However, this T should also not be too large. Otherwise, the calculated program build speed will not closely reflect the actual program build speed. In our implementation, T is chosen to be 5. In our experiments, we found that this number is sufficient to provide a smooth estimate of the current program build speed.

This approach to calculating the conversion from U to time is admittedly simplistic, and although it worked well in our experiments, there are cases in which it will be misleading. For example, a problem arises when two source code files that are of the same size have very different values of the compile cost, c_i, due to radically different program structures. This problem could be alleviated by a more complex modeling of the compile cost—ideally this modeling should take into account both the size and the program structure of the source code file.

4.4. Handling different combinations of optimization flags

The above discussion assumes that all programs are built using the same optimization flags. In practice, users can change optimization flags arbitrarily. Different values of K_c, L_c, K_l and L_l need to be used for different combinations of optimization flags, as the program build cost depends heavily on the optimization flags used. As a result, the challenge for us is to handle all possible combinations of optimization flags without excessive storage and computation overhead.

4.4.1. Non-compression method

A simple method is to build a large table, T_{large}, for the four factors K_c, L_c, K_l, and L_l. For each combination of optimization flags, there is a row in the table T_{large} that records the corresponding initial values of the four factors K_c, L_c, K_l, and L_l—the program compilation tool computes these

DOI: 10.1002/spe

initial values by using this specific combination of optimization flags to build the k training programs. Then each time the combination of optimization flags changes (or if different source code files in the same program are compiled using different combinations of optimization flags), we go to T_{large} to find the appropriate values of the four factors K_c, L_c, K_l, and L_l. We call this simple method the *non-compression method*.

However, this method requires excessive storage and computation overhead and hence is not feasible in practice. For example, GCC [25] has more than 40 optimization flags, which can produce more than $2^{40} \approx 10^{12}$ different combinations. In the following section, we describe a method that does not incur either excessive storage overhead or excessive computation overhead.

4.4.2. *Main idea of the compression method*

We call our method the *compression method*, as the main idea of our method is to 'compress' T_{large} (the large table used in the non-compression method) into a small table, T_{small}. More specifically, we find some heuristic formulas that can approximately describe the relationship among different optimization flags. We store the factor values for a few different combinations of optimization flags that are deemed to be the most critical. Then for each combination of optimization flags, the appropriate factor values are computed using both the heuristic formulas and the information stored in the small table T_{small}.

We first describe our techniques for computing the two factors related to compiling, K_c and L_c. Then we discuss how to compute the two factors related to linking, K_l and L_l.

4.4.3. *Small table (T_{small})*

In general, a compiler has multiple (say, h) general optimization levels that are most frequently used in practice. The lowest optimization level 0 does not perform any optimization (or only performs a minimal optimization). Each other optimization level i $(1 \leq i \leq h-1)$ turns on a few optimization flags in addition to those optimization flags that have already been turned on at the optimization level $i-1$.

We keep a small table T_{small}. For each optimization level i $(0 \leq i \leq h-1)$, there is a row in the table T_{small} that records the corresponding initial values of the two factors K_c and L_c—the program compilation tool computes these initial values by using optimization level i to build the k training programs. Whenever a program is built using optimization level i $(0 \leq i \leq h-1)$, we find the values of the two factors K_c and L_c that correspond to optimization level i. The techniques in Section 4.2 are used to both continuously refine the estimated program build cost and update the values of the two factors K_c and L_c for optimization level i. In this way, we can handle the most common cases—the h optimization levels.

4.4.4. *Heuristic formulas*

To handle other combinations of optimization flags, we use heuristic formulas to explore the relationship among different optimization flags. Note that in this case, we only use the information stored in the small table T_{small}. We do not update the information stored in T_{small}, as no single row in T_{small} corresponds to the particular combination of optimization flags.

DOI: 10.1002/spe

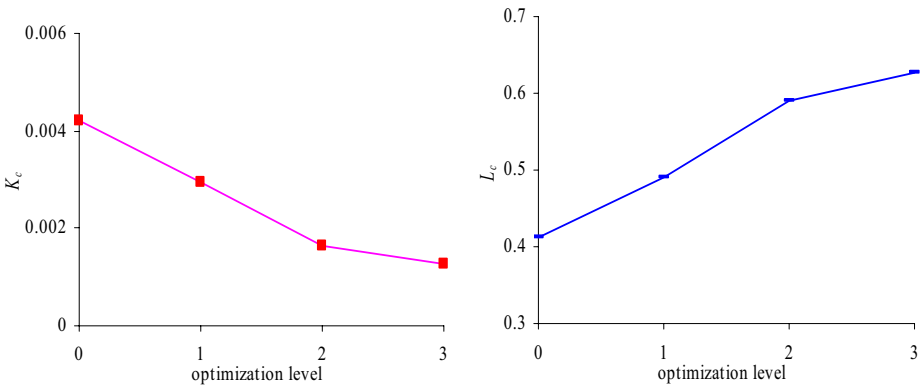

Figure 3. (a) K_c versus optimization level. (b) L_c versus optimization level.

To motivate these heuristic formulas, we first examine a typical set of K_c's and L_c's that we computed for GCC [25] for all the optimization levels (the experiment environment is described in detail in Section 5). Figure 3(a) shows the relationship between K_c and the optimization level. Figure 3(b) shows the relationship between L_c and the optimization level. We can see that a general trend exists: as the optimization level increases, K_c decreases while L_c increases. In other words, as more optimization flags are turned on, K_c decreases while L_c increases. This monotonicity suggests that we can use heuristic formulas that are of additive form to compute the corresponding K_c and L_c terms for arbitrary combinations of optimization flags.

In general, all optimization flags can be classified into two categories.

- *High-impact flag*. The addition of a high-impact flag may have a large effect on the program build cost. This kind of flags include loop unroll, function inline, and inter-procedural analysis.
- *Low-impact flag*. The addition of a low-impact optimization flag has only a small effect on the program build cost. This kind of flags include dead code elimination, sparse conditional constant propagation, and strength reduction.

For simplicity, we assume that the effect on K_c (L_c) from different optimization flags is additive. All low-impact flags have the same effect on K_c (L_c). Note that in reality, different low-impact flags have different degrees of effect on K_c (L_c). Hence, assuming that all low-impact flags have the same effect on K_c (L_c) will introduce errors into the estimates provided by the PIs. However, as each low-impact flag has only a small effect on the program build cost, this assumption will only introduce a small error for an individual low-impact flag. If a small number of low-impact flags are turned on, the combined error introduced by this assumption for all these flags is small. If a large number of low-impact flags are turned on, the errors introduced by this assumption for different low-impact flags, which can be either positive or negative, are likely to cancel each other out. Then the combined error introduced by this assumption for all these flags will still be small. Therefore, in practice, we would expect this assumption to work reasonably well in most cases.

DOI: 10.1002/spe

High-impact flags are treated separately, as each high-impact flag can have a large effect on the program build cost. Whenever possible, (piecewise) linear interpolation is used. This leads to our heuristic formulas. In Section 5.5, we show that these heuristic formulas work well.

In the rest of this section, among all the high-impact flags, only the function inline flag is discussed. The loop unroll flag and inter-procedural analysis can be handled in a similar way. The compiler has a parameter p_f that gives the threshold for enabling/disabling function inlining (e.g. when the size of a function that can be inlined is smaller than p_f, inlining is applied to this function). Let K_c^l (L_c^l) denote the value of K_c (L_c) at the lowest optimization level 0. Let $K_c^{(0)}$ ($L_c^{(0)}$) denote the value of K_c (L_c) when all low-impact flags are turned on. A large number G is chosen such that users will typically not let p_f exceed G. There is a step size g. For each i ($1 \leq i \leq G/g + 1$), let $K_c^{(i)}$ ($L_c^{(i)}$) denote the value of K_c (L_c) when all low-impact flags and the function inline flag are turned on, and $p_f = i \times g$.

For each set of optimization flags S_f (and the associated parameter p_f) mentioned in the previous paragraph, the program compilation tool computes the corresponding initial values of K_c and L_c by using S_f to build the k training programs. These initial values are stored in the small table T_{small}.

Suppose there are M low-impact optimization flags in total. In the case that m ($1 \leq m \leq M$) low-impact optimization flags are turned on, the corresponding K_c and L_c are computed using the following heuristic formulas:

$$K_c = K_c^l + (K_c^{(0)} - K_c^l) \times m/M, \quad L_c = L_c^l + (L_c^{(0)} - L_c^l) \times m/M$$

These two formulas essentially perform a linear interpolation between the factor value at optimization level 0 and the factor value when all low-impact flags are turned on. In the case that both m ($1 \leq m \leq M$) low-impact optimization flags and the function inline flag are turned on, and the parameter value associated with the function inline flag is p_f, the corresponding K_c and L_c are computed using the following heuristic formulas:

$$K_c = K_c^l + (K_c^{(0)} - K_c^l) \times m/M + (K_c^{(j)} - K_c^{(0)}) + (K_c^{(j+1)} - K_c^{(j)}) \times (p_f - j \times g)/g$$
$$L_c = L_c^l + (L_c^{(0)} - L_c^l) \times m/M + (L_c^{(j)} - L_c^{(0)}) + (L_c^{(j+1)} - L_c^{(j)}) \times (p_f - j \times g)/g$$

where $j = \lfloor p_f/g \rfloor$, and $\lfloor x \rfloor$ is the floor function. These two formulas essentially perform a linear interpolation between the factor value at optimization level 0 and the factor value when all low-impact flags are turned on, and a piecewise linear interpolation for the parameter p_f. Note that if $p_f > G$, which should rarely happen as we choose G large enough so that users will typically not let p_f exceed G, we replace p_f with G in the above computation. In our implementation, we choose $g = 500$ and $G = 10\,000$. In our experiments, we found that these two numbers work well.

So far, only the two factors K_c and L_c have been discussed. The other two factors K_l and L_l exhibit similar behavior: as more optimization flags are turned on, K_l decreases while L_l increases. For example, consider a typical set of K_l and L_l terms that we computed for GCC (the experimental environment is described in detail in Section 5 below). Figure 4(a) shows the relationship between K_l and the optimization level, and Figure 4(b) shows the relationship between L_l and the optimization level. We can see that the general trend similar to that of K_c and L_c does exist: as the optimization level increases, K_l decreases while L_l increases. Due to this similarity, we use the same techniques as used for K_c and L_c to handle K_l and L_l.

DOI: 10.1002/spe

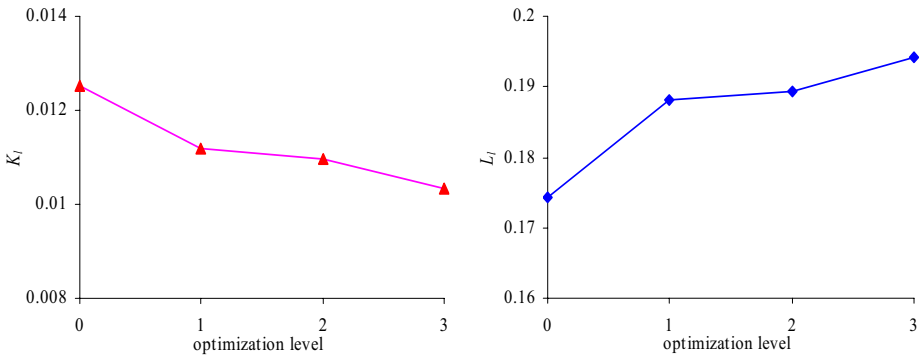

Figure 4. (a) K_l versus optimization level. (b) L_l versus optimization level.

5. PERFORMANCE EVALUATION

In this section, we present the performance results of PIs for program compilation. We augmented GNU Make [13,14] with a prototype implementation of PIs. Although any OS commands can be invoked from Make, in our experiments, we focus on the commands directly used (no aliasing) for compiling and linking programs. We leave it as an interesting area for future work to explore how to support PIs in the presence of other commands. In all our experiments, our prototyped PIs could be updated every second with negligible overhead and provide useful information, which we consider to have met the four goals mentioned in Section 3: continuously revised estimates, acceptable pacing, minimal overhead, and maximal functionality.

5.1. Experiment description

We first present the experiment results for GCC. Our measurements were performed with the GNU Make running on a Dell Inspiron 4000 PC with a 600 MHz Intel Pentium III processor, 512 MB main memory, a 40 GB IDE disk, and running the Linux release 2.6.9-1.667 operating system. GCC version 3.4.2 [25] was used.

We compared the modeling cost method, which is proposed in Section 4, with the *counting file method*. The counting file method is the simple method mentioned in the introduction. First, it counts the number of source code files compiled so far. Then it reports the percentage of the program build task that has been completed as the ratio of the number of compiled source code files to the total number of source code files that need to be compiled.

An additional feature is added to the original counting file method, as that method does not provide estimate of the remaining program build time. The current program build speed is estimated as the average number of source code files compiled per second in the last $T = 5$ s. At any time before linking is started, the remaining program build time is estimated to be the ratio of the number of source code

DOI: 10.1002/spe

files remaining to be compiled to the observed current program build speed. Once all the source code files have been compiled and before linking is started, the remaining program build time is reported to be zero, since we have no estimate of the link time.

We used the 11 C programs in the SPEC CINT2000 benchmark suite: *gzip*, *vpr*, *gcc*, *mcf*, *crafty*, *parser*, *perlbmk*, *gap*, *vortex*, *bzip2*, and *twolf* [26]. Among these 11 programs, we chose the program *gcc*, which has the longest build time, as the test program. This test program was used to evaluate the performance of the PIs. (We also tested other programs but the results were similar and are thus omitted here.) In the modeling cost method, the other $k = 10$ programs were used as training programs to compute the initial values of the four factors K_c, L_c, K_l, and L_l. In all experiments, the outputs of PIs were stored into a file.

For GCC, five kinds of experiments were performed to evaluate the performance of PIs. The first experiment (the existing flag experiment) shows that the modeling cost method is effective and better than the counting file method. The second experiment (the repeated building experiment) shows that the modeling cost method can adapt to the program compilation tool's estimation errors. The third experiment (the workload interference experiment) shows that the modeling cost method can adapt to varying run-time system loads. The fourth experiment (the random flag experiment) shows that the compression method works well for different combinations of optimization flags. Finally, the fifth experiment (the PostgreSQL experiment) shows that the modeling cost model works well for very large programs.

5.2. Existing flag experiment

In this experiment, the −O3 flag was used to build the *gcc* test program on an unloaded system. Note that for the −O3 flag (optimization level 3), there is a corresponding row in the small table, T_{small}. The purpose of this experiment is to show that the modeling cost method is effective and better than the counting file method.

We first investigate the relationship between the compile/link cost and the file size in the modeling cost method. This is to see how well our two heuristic cost estimation formulas $c_i = K_c \times s_i^{L_c}$ and $c_{\text{link}} = K_l \times S^{L_l}$ model reality.

Figure 5(a) is the scatter plot that shows the relationship between the measured compile cost c_i and the file size s_i for all the source code files in the 11 programs (one test program and 10 training programs). Figure 5(b) is the scatter plot that shows the relationship between the measured link cost c_{link} and the total file size $S = \sum_{i=1}^{n} s_i$ for all the 11 programs. In both Figure 5(a) and (b), a logarithmic scale is used for both the x-axis and the y-axis, and the dotted line is the regression line.

We can see that $\ln c_i$ and $\ln s_i$ approximately have a linear relationship. Hence, the heuristic formula $c_i = K_c \times s_i^{L_c}$ roughly reflects the relationship between the compile cost c_i and the file size s_i. Similarly, $\ln c_{\text{link}}$ and $\ln S$ approximately have a linear relationship. Hence, the heuristic formula $c_{\text{link}} = K_l \times S^{L_l}$ roughly reflects the relationship between the link cost c_{link} and the total file size S.

Now we describe the performance of PIs using the *gcc* test program. Figure 6(a) shows the program build cost estimated by the modeling cost method over time, with the exact program build cost indicated by the horizontal dotted line. During the entire execution of the program build task, the program build cost estimated by the modeling cost method is not far from the exact program build cost. This shows that the cost estimation techniques used in the modeling cost method are quite effective. As the *gcc* test program is being built, the program build cost estimated by the modeling cost method keeps

DOI: 10.1002/spe

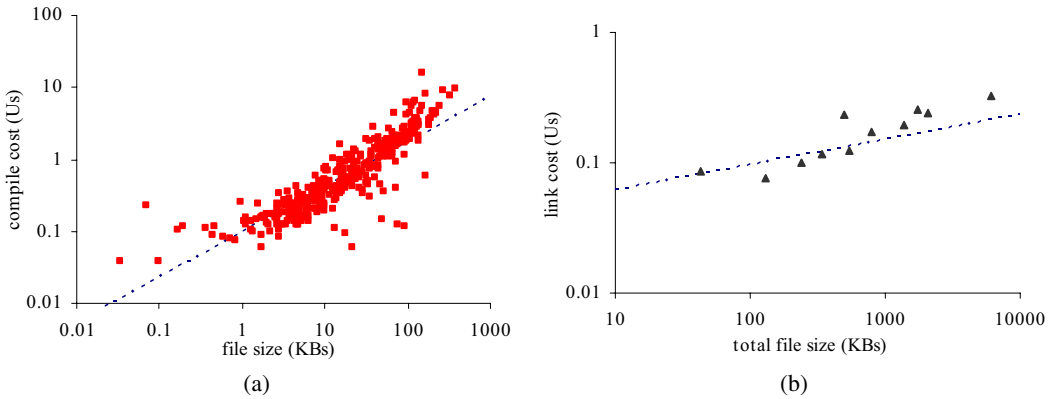

Figure 5. (a) Compile cost versus file size. (b) Link cost versus total file size.

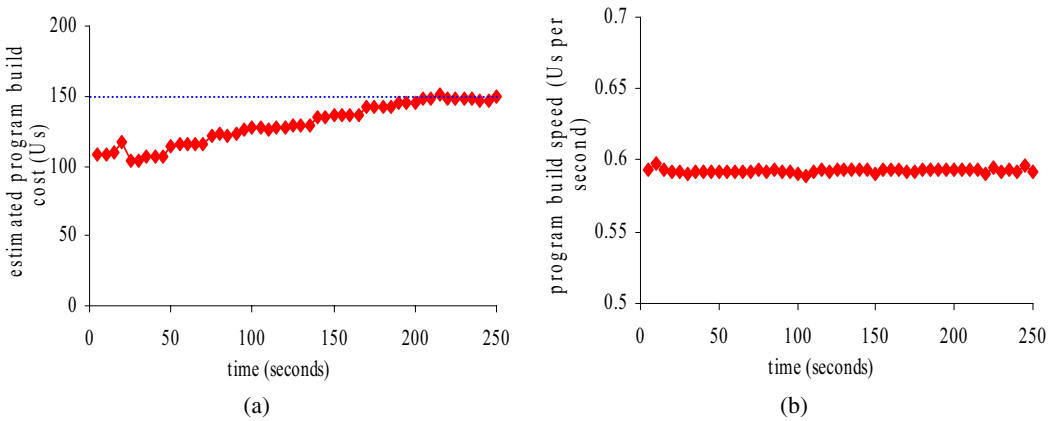

Figure 6. (a) Program build cost estimated over time (modeling cost method). (b) Program build speed monitored over time (modeling cost method).

approaching the exact program build cost. When the program build task finishes, the program build cost estimated by the modeling cost method reaches the exact program build cost.

Figure 6(b) shows the program build speed monitored by the PI over time in the modeling cost method. The program build task is the only task that runs in the system. Hence, during the entire execution of the program build task, the monitored program build speed in the modeling cost method is quite stable.

DOI: 10.1002/spe

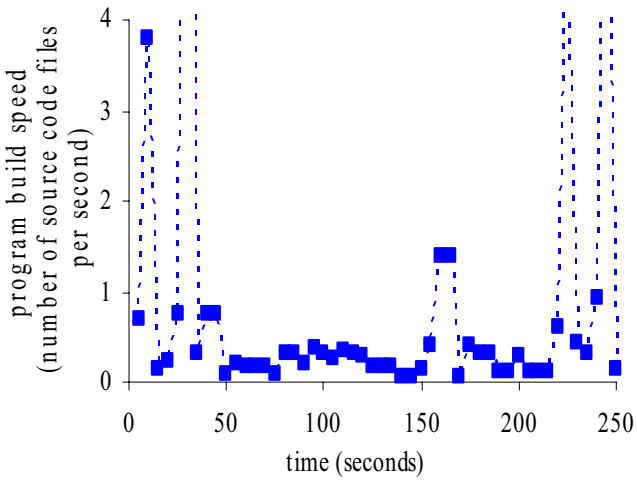

Figure 7. Program build speed monitored over time (counting file method).

The counting file method uses the number of source code files in the program to estimate the program build cost. Hence, during the entire execution of the program build task, the program build cost estimated by the counting file method remains as a constant.

Figure 7 shows the program build speed monitored by the PI over time in the counting file method. Even in the case that the program build task is the only task running in the system, the number of source code files compiled per second (which has already been smoothed by averaging over the last $T = 5$ s) still varies significantly from time to time. Hence, using the number of source code files to represent the program build cost is not a good choice.

Figure 8 shows the remaining program build time estimated by the PI over time, with the actual remaining program build time indicated by the dashed line. During the entire execution of the program build task, the remaining program build time estimated by the modeling cost method is fairly close to the actual remaining program build time. The closer to the completion of the program build task, the more precise the remaining program build time estimated by the modeling cost method. In contrast, the remaining program build time estimated by the counting file method can oscillate greatly over time and often differs from the actual remaining program build time by several times (either overestimate or underestimate). This is because in the modeling cost method, both the estimated program build cost and the monitored program build speed remain quite stable during the entire execution of the program build task. The closer to the completion of the program build task, the more precise the program build cost estimated by the modeling cost method. In contrast, in the counting file method, the monitored program build speed varies significantly from time to time, while the estimated program build cost remains as a constant.

In summary, compared to the counting file method, the modeling cost method can provide more precise and more smoothly updated estimates. Also, the estimates provided by the modeling cost method are fairly close to the real values. Hence, the modeling cost method can be regarded as better

DOI: 10.1002/spe

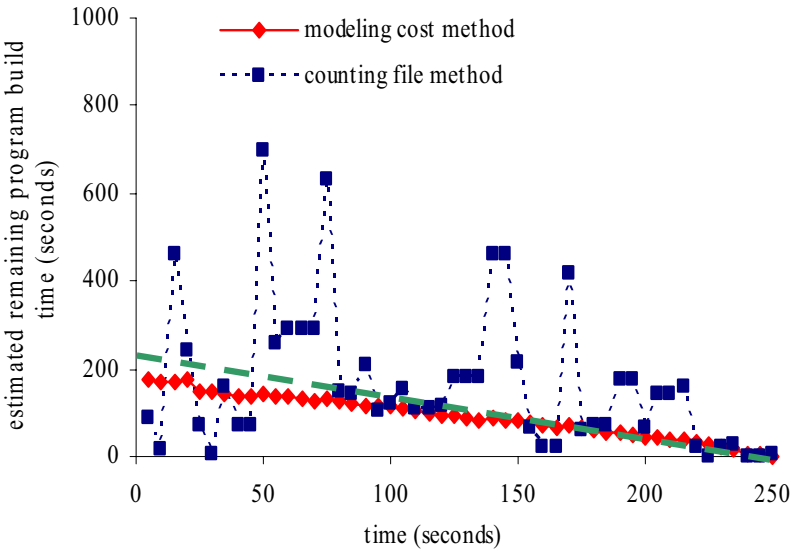

Figure 8. Remaining program build time estimated over time (existing flag experiment).

than the counting file method. In the rest of Section 5, we no longer present the experiment results of the counting file method. Rather, we only present the experiment results of the modeling cost method.

In the existing flag experiment, in the case that the PI is updated every second, the overhead of the modeling cost method is 0.36 s. In contrast, building the *gcc* test program takes 250 s. Hence, our PIs have negligible overhead ($0.36/250 = 0.14\%$).

5.3. Repeated building experiment

In this experiment, the $-$O3 flag was used to build the *gcc* test program on an unloaded system twice. On both occasions we started from scratch and built the entire *gcc* test program. The purpose of this experiment is to show how our PI adjusts to the program compilation tool's estimation errors for the modeling cost method.

Figure 9 shows the program build cost estimated by the modeling cost method over time, with the exact program build cost indicated by the horizontal dotted line. Initially, the four factors K_c, L_c, K_l, andL_l are computed using the 10 training programs and do not fit well with the *gcc* test program. After the *gcc* test program has been built once, the values of the four factors K_c, L_c, K_l, andL_l fit better with the *gcc* test program. Hence, compared to the first build of the *gcc* test program, the modeling cost method estimates the program build cost more precisely on the second build.

Figure 10 shows the remaining program build time estimated by the modeling cost method over time, with the actual remaining program build time indicated by the dashed line. Compared to the first build,

DOI: 10.1002/spe

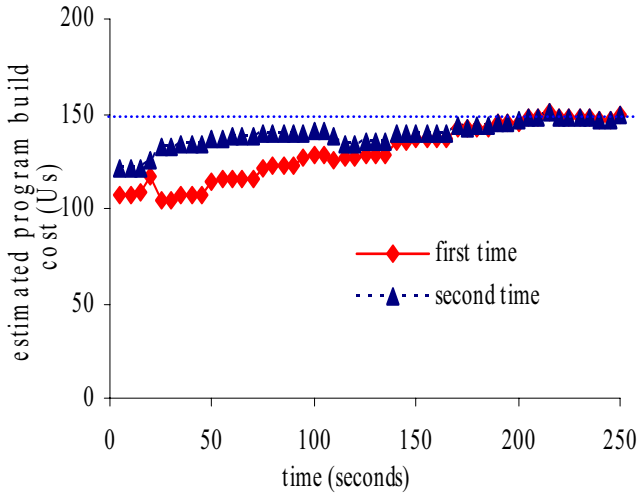

Figure 9. Program build cost estimated over time (repeated building experiment).

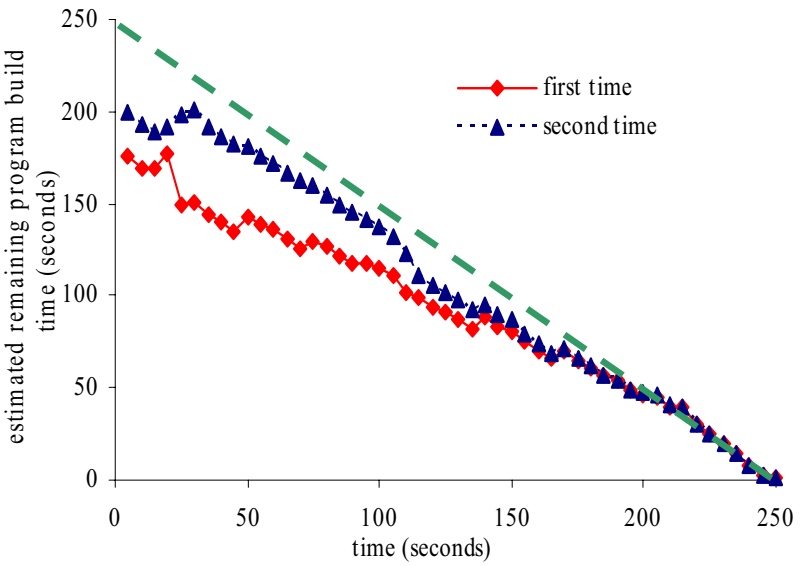

Figure 10. Remaining program build time estimated over time (repeated building experiment).

DOI: 10.1002/spe

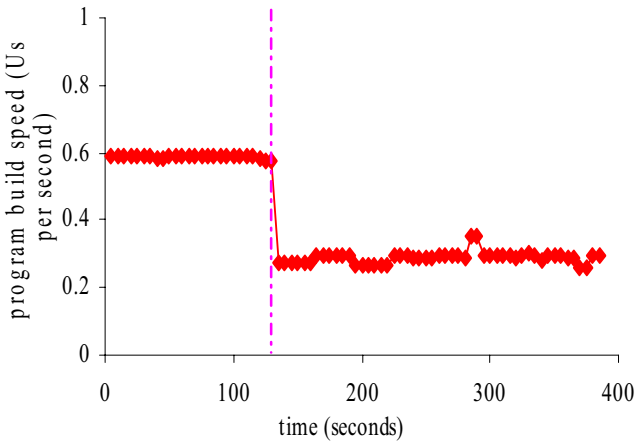

Figure 11. Program build speed monitored over time (workload interference experiment).

the modeling cost method estimates the remaining program build time more precisely on the second build. This is due to the same reason as previously explained.

5.4. Workload interference experiment

In this experiment, the −O3 flag was used to build the *gcc* test program. We started the program build task on an unloaded system. Without any interruption, this program build task will run for 250 s. In the middle of the execution of the program build task (at 120 s), a CPU-intensive program was started. This CPU-intensive program kept running until the *gcc* program build task finished execution. The purpose of this experiment is to show how our PI adjusts to varying run-time system loads for the modeling cost method. In each figure of Section 5.4, a vertical dash-dotted line is used to represent the time when the CPU-intensive program starts execution.

Figure 11 shows the program build speed monitored by the PI over time in the modeling cost method. Before the CPU-intensive program starts execution, the *gcc* test program is built at the same speed as that in the existing flag experiment (Figure 6(b)). However, once the CPU-intensive program starts execution, the build speed of the *gcc* test program is decreased (almost by 50%). This situation continues throughout the remaining execution of the *gcc* program build task.

Figure 12 shows the remaining program build time estimated by the modeling cost method over time, with the actual remaining program build time indicated by the dashed line. Before the CPU-intensive program starts execution, the shape of the curve in Figure 12 is similar to the shape of the corresponding curve in Figure 8. At 120 s, due to the start of the CPU-intensive program, the remaining program build time estimated by the modeling cost method increases sharply. After 120 s, the dashed line is fairly close to the curve that represents the remaining program build time estimated by the modeling cost method. That is, after 120 s, the remaining program build time that is estimated by the modeling cost method is fairly precise.

DOI: 10.1002/spe

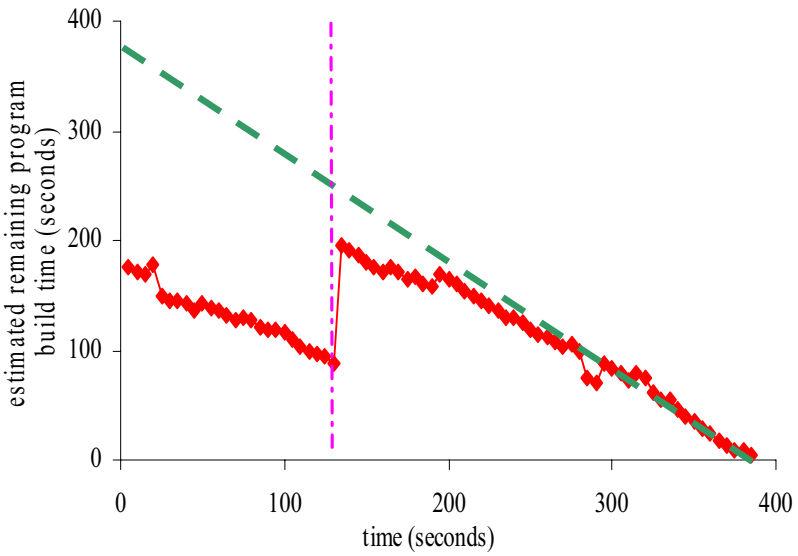

Figure 12. Remaining program build time estimated over time (workload interference experiment).

5.5. Random flag experiment

In this experiment, the *gcc* test program was built on an unloaded system one hundred times (100 test runs). Each time a random number of optimization flags were turned on. In the case that some of the high-impact flags were turned on, random values were given to their associated parameters. We compared the compression method with the non-compression method. In the non-compression method, for each combination of optimization flags, the initial values of K_c, L_c, K_l, and L_l were obtained by actually building the $k = 10$ training programs.

As shown in Section 5.2, the modeling cost method usually has the greatest estimation error at the beginning of building the *gcc* test program. Also, the errors of the estimates provided by the modeling cost method mainly come from the error of the estimated program build cost. (The monitored program build speed is quite stable during the entire execution of the program build task.) Hence, when comparing the compression method with the non-compression method, we use the cost estimation error at the beginning of building the *gcc* test program as the performance metric.

Suppose the actual program build cost is c_{actual}. At the beginning of building the *gcc* test program, the compression method estimates the program build cost to be $c_{\text{compression}}$. We define the relative error of the program build cost estimated by the compression method as $|c_{\text{compression}} - c_{\text{actual}}|/c_{\text{actual}} \times 100\%$. We define the relative error of the program build cost estimated by the non-compression method in a similar way.

DOI: 10.1002/spe

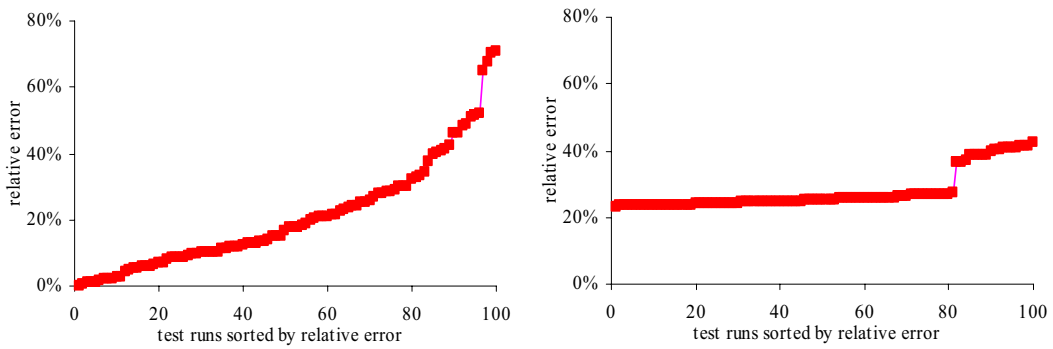

Figure 13. Relative error of the program build cost: (a) estimated using the compression method; (b) estimated using the non-compression method.

Figure 13(a) shows the relative error of the program build cost estimated by the compression method in the 100 test runs, where all test runs are sorted in ascending order of the relative error. Figure 13(b) shows the corresponding case for the non-compression method.

Compared to that of the compression method, the relative error of the estimated program build cost of the non-compression method is more stable over the 100 test runs. This is because the compression method makes some simplified assumptions. These assumptions do not match exactly with reality and thus introduce some randomness into the error of the estimated program build cost (the error is increased in some cases and decreased in other cases). However, on average, the compression method performs no worse than the non-compression method. In all the 100 test runs, the maximum relative error of the estimated program build cost of the compression method (71%) is no more than two times worse than that of the non-compression method (42%). Also, it is relatively small compared to the degree (six times, or 600%) to which the actual program build cost can vary in all the 100 test runs. Hence, compared to the non-compression method, the compression method is more desirable, as it achieves comparable performance with a much smaller storage and computation overhead.

5.6. PostgreSQL experiment

In this experiment, the *gcc* test program was replaced with the *PostgreSQL* software Version 8.1.4 [27]. *PostgreSQL* is a widely used open-source RDBMS. Its code size is about five times larger than that of *gcc*. The results for *PostgreSQL* are similar to those for *gcc*. We present the result of the estimated remaining program build time when the −O3 flag was used to build the *PostgreSQL* test program on an unloaded system. Figure 14 shows the remaining program build time estimated by the modeling cost method over time, with the actual remaining program build time indicated by the dashed line. The shape of the curve in Figure 14 is similar to the shape of the corresponding curve in Figure 8.

DOI: 10.1002/spe

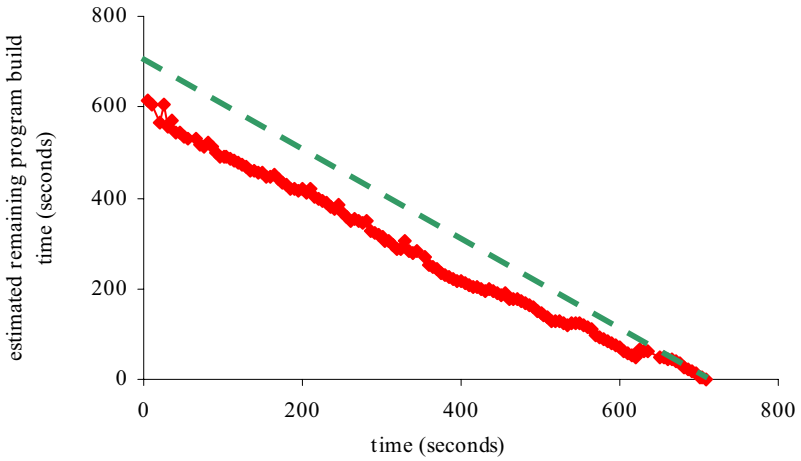

Figure 14. Remaining program build time estimated over time (PostgreSQL).

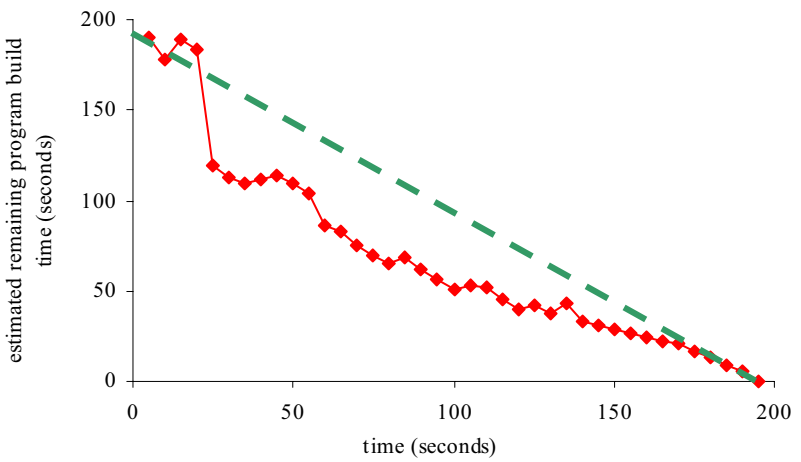

Figure 15. Remaining program build time estimated over time (Fortran).

DOI: 10.1002/spe

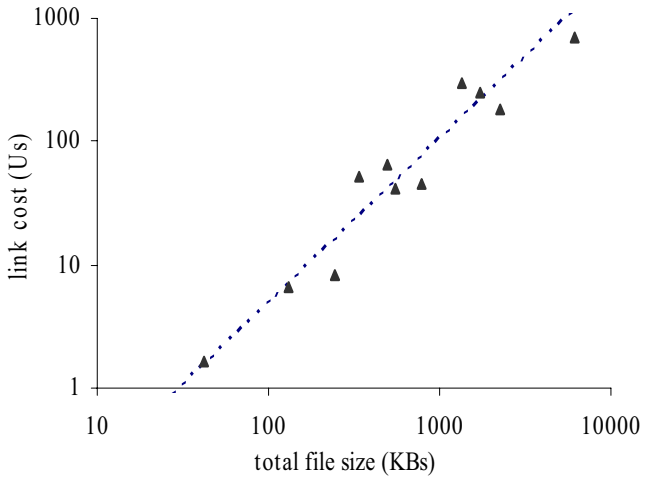

Figure 16. Link cost versus total file size (XLC–05).

5.7. Fortran77 experiment results

In order to verify the generality of our approach, experiments were also conducted on some other compilers and computers. The results in all cases show that our modeling cost method combined with the compression method can let PIs provide good estimates. In the following subsections, for these experiments, we either present a representative set of experiment results or only point out the major observation that differ from the GCC experiment results. The other detailed experiment results are omitted.

We first replaced GCC with G77 version 3.4.2 [28], a Fortran77 compiler. We used the six Fortran77 programs in the SPEC CFP2000 benchmark suite: *wupwise*, *swim*, *mgrid*, *applu*, *sixtrack*, and *apsi* [26]. Among these six programs, we chose the program *sixtrack*, which has the longest build time, as the test program. This test program was used to evaluate the performance of PIs. In the modeling cost method, the other $k = 5$ programs were used as training programs to compute the initial values of the four factors K_c, L_c, K_l, and L_l. The results for the Fortran77 programs are similar to those for the C programs. We present the result of the estimated remaining program build time when the −O3 flag was used to build the *sixtrack* test program on an unloaded system. Figure 15 shows the remaining program build time estimated by the modeling cost method over time, with the actual remaining program build time indicated by the dashed line. The shape of the curve in Figure 15 is similar to the shape of the corresponding curve in Figure 8.

5.8. XLC experiment results

We used GNU Make and IBM XLC compiler version 6.0 [29] to build the 11 C programs in the SPEC CINT 2000 benchmark suite on an IBM SP2 computer with four 375 MHz IBM Power3 processors,

DOI: 10.1002/spe

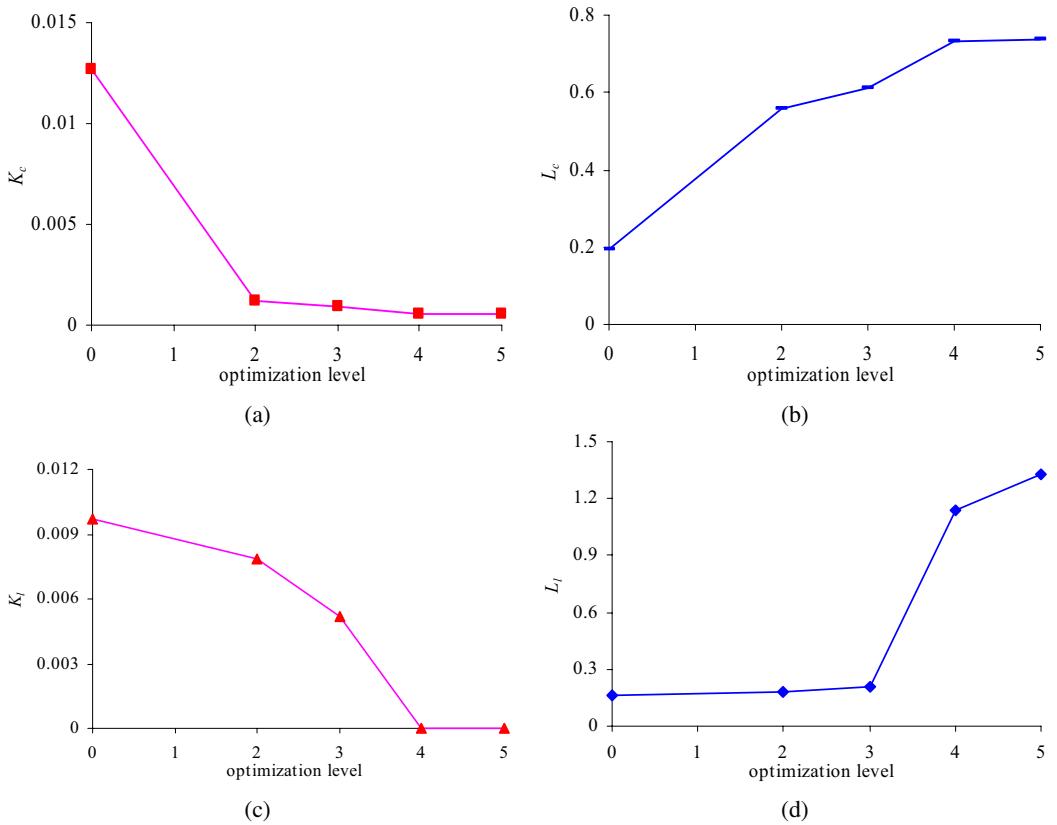

Figure 17. (a) K_c versus optimization level (XLC). (b) L_c versus optimization level (XLC). (c) K_l versus optimization level (XLC). (d) L_l versus optimization level (XLC).

2 GB main memory, two 18 GB SCSI disks, and running the AIX 5.1 operating system. XLC has five optimization levels: 0, 2, 3, 4, and 5. Note that there is no optimization level 1. At optimization levels 4 and 5, XLC invokes inter-procedural analysis, which is never used in GCC.

As shown in Figure 5(b), using GCC to link a program usually takes less than one second and is fairly cheap. This is also the case with XLC when inter-procedural analysis is disabled. However, in XLC, once inter-procedural analysis is invoked, the link cost suddenly becomes much more expensive (say, hundreds of times), as XLC puts most operations of inter-procedural analysis into the linking step (rather than into the compiling step).

Figure 16 is the scatter plot that shows the relationship between the measured link cost c_{link} and the total file size $S = \sum_{i=1}^{n} s_i$ for all the 11 programs, where XLC was used to build the 11 programs with the –O5 flag (optimization level 5). In Figure 16, a logarithmic scale is used for both the x-axis and the y-axis, and the dotted line is the regression line. We can see that even in the case of a fairly

DOI: 10.1002/spe

expensive link cost (as inter-procedural analysis is invoked at optimization level 5), ln c_{link} and ln S still roughly have a linear relationship. That is, our heuristic cost estimation formula $c_{\text{link}} = K_l \times S^{L_l}$ still models reality well. (Recall that Figure 5(b) shows that in the case of cheap link cost, ln c_{link} and ln S approximately have a linear relationship.)

For XLC, we show the relationship between K_c, L_c, K_l, and L_l and the optimization level in Figures 17(a)–(d). There, the four factors K_c, L_c, K_l, and L_l exhibit the same trend as that in the case of GCC: as the optimization level increases, K_c and K_l decrease while L_c and L_l increase. Note that this trend persists even after inter-procedural analysis is invoked at optimization levels 4 and 5, and the link cost becomes fairly expensive. As discussed in Section 4.4, this trend is crucial for the compression method to work well.

6. CONCLUSION

In this paper, we have proposed techniques for supporting PIs for program compilation. Our main idea is that as a large program is being built, we continuously refine the program build cost estimate and monitor the current program build speed. Then we continuously give the user an estimate of both the percentage of the program build task that has been completed and the remaining program build time. Our experiments show that a PI based upon our techniques is useful for program compilation, and that it adapts both to the program compilation tool's estimation errors and to varying run-time system loads. Since our techniques can estimate the program build time, they can also be used to compare the (program building) efficiency of different compilers.

In RDBMSs, PIs have been used to facilitate workload management [30]. There, the execution cost of a fixed SQL query is a constant. In program compilation, the program built time can be changed by adjusting the level of optimization. It would be interesting to investigate whether the method proposed in Luo *et al*. [30] can be modified to be used in program compilation.

REFERENCES

1. Myers BA. The importance of percent-done progress indicators for computer-human interfaces. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, San Francisco, CA, 1985. ACM Press: New York, 1985; 11–17.
2. Landay J. Lecture Notes on User Interface Design, Prototyping, and Evaluation. http://guir.berkeley.edu/courses/cs160/2002_spring/lectures_files/heuristic-evaluation.pdf [2004].
3. Berque DA, Goldberg MK. Monitoring an algorithm's execution. *Computational Support for Discrete Mathematics* (*DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 15), Dean N, Shannon GE (eds.). American Mathematical Society: Providence, RI, 1992; 153–163.
4. Chaudhuri S, Narasayya VR, Ramamurthy R. Estimating progress of long running SQL queries. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France, 13–18 June 2004. ACM Press: New York, 2004; 803–814.
5. Lohman G. Personal communication, 2005.
6. Compile Time Stats. http://linuxreviews.org/gentoo/compiletimes [2005].
7. Kvitka C. Tapping into the Grid. http://www.oracle.com/technology/oramag/oracle/02-nov/o62news.html [2004].
8. Building an Executable Application. http://www.functionalobjects.com/products/doc/env/env_15.htm [2004].
9. Emerge Progress Bar. http://forums.gentoo.org/viewtopic.php?t=42346&highlight=emergeprogress&sid=3c66f1e0ff64d1c988e565486c467cae [2004].
10. Installing Mathematica Fonts for Exceed v 7.1. http://groups.haas.berkeley.edu/HCS/howdoi/Installing_Mathematica_Fonts_for_Exceed_v7.pdf [2004].

DOI: 10.1002/spe

11. Make Progress Bar. http://www.ces.clemson.edu/~eduffy/prog.png [2004].
12. Shaw WH, Howatt JW, Maness RS, Miller DM. A software science model of compile time. *IEEE Transactions on Software Engineering* 1989; **15**(5):543–549.
13. Feldman SI. Make—a program for maintaining computer programs. *Software—Practice and Experience* 1979; **9**(4): 255–65.
14. GNU Make. http://www.gnu.org/software/make [2005].
15. Braun JL. Dynamic control of compile time using vertical region-based compilation. *MS Thesis*, University of Illinois at Urbana-Champaign, IL, 1998.
16. Flajolet P, Steyaert J. A complexity calculus for recursive tree algorithms. *Mathematical Systems Theory* 1987; **19**(4): 301–331.
17. Hickey TJ, Cohen J. Automating program analysis. *Journal of the ACM* 1988; **35**(1):185–220.
18. Sarkar V. Determining average program execution times and their variance. *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, 21–23 June 1989. ACM Press: New York, 1989; 298–312.
19. Wegbreit B. Mechanical program analysis. *Communications of the ACM* 1975; **18**(9):528–539.
20. Luo G, Naughton JF, Ellmann CJ. Watzke M. Toward a Progress Indicator for Database Queries. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France 13–18 June 2004. ACM Press: New York, 2004; 791–802.
21. Ramakrishnan R, Gehrke JE. *Database Management Systems* (3rd edn). McGraw-Hill: New York, 2002.
22. Ilyas IF, Rao J, Lohman GM. Gao D, Lin E. Estimating compilation time of a query optimizer. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, CA, 9–12 June 2003. ACM Press: New York, 2003; 373–384.
23. Selinger PG, Astrahan MM, Chamberlin DD. Lorie RA, Price TG. Access path selection in a relational database management system. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, Boston, MA, 30 May–1 June 1979. ACM Press: New York, 1979; 23–34.
24. Bickel PJ, Doksum KA. *Mathematical Statistics: Basic Ideas and Selected Topics*, vol. 1. Prentice-Hall: Englewood Cliffs, NJ, 2001.
25. GCC Homepage. http://gcc.gnu.org [2005].
26. SPEC Benchmark Homepage. http://www.spec.org [2005].
27. PostgreSQL Homepage. http://www.postgresql.org [2006].
28. G77 Homepage. http://www.gnu.org/software/fortran/fortran.html [2005].
29. IBM XL C Compiler Homepage. http://www-306.ibm.com/software/awdtools/ccompilers [2005].
30. Luo G, Naughton JF, Yu PS. Multi-query SQL progress indicators. *Proceedings of the 2006 International Conference on Extending Database Technology (EDBT'06)*, Munich, Germany, March 2006 (*Lecture Notes in Computer Science*, vol. 3896). Springer: Berlin, 2006; 921–941.

DOI: 10.1002/spe