Automatic Parallelisation of Web Applications

Gian Perrone
Department of Computer Science
University of Waikato
Hamilton, New Zealand
gdp3@cs.waikato.ac.nz

David Streader
Department of Computer Science
University of Waikato
Hamilton, New Zealand
dstr@cs.waikato.ac.nz

ABSTRACT

Small web applications have a tendency to get bigger. Yet despite the current popularity of web applications, little has been done to help programmers to leverage the performance and scalability benefits that can result from the introduction of parallelism into a program. Accordingly, we present a technique for the automatic parallelisation of whole web applications, including persistent data storage mechanisms. We detail our prototype implementation of this technique, *Ceth* and finally, we establish the soundness of the process by which we extract coarse-grained parallelism from programs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution—*Conversion from sequential to parallel forms*

Keywords

Parallelising compilers, parallelisation, web applications

1. INTRODUCTION

The ever-increasing amount of parallel hardware being incorporated into commodity computer systems is driving the development of tools designed to enable programmers to take advantage of these advances in hardware without introducing extra complexity into the development cycle, or requiring legacy code to be discarded. The way in which this parallel hardware is programmed is very different to the way in which conventional "single-threaded" hardware is programmed. Without changes in the way software is developed, parallel hardware will not necessarily be used to its full potential. To this end, it remains an open question as to how we can continue to develop software using existing methods, while making greater use of parallelism.

While there have been attempts to provide general solutions to this problem, the survey by Mukherjee and Gurd [12] suggests that such "one-size-fits-all" approaches have not yet yielded the results or simplicity that might allow them to be widely adopted. We propose a compilation and analysis system "Ceth", which programmatically extracts course-grained parallelism from sequential, imperative programs. Ceth is designed specifically for the web application domain (i.e. it deliberately attempts to solve only a subset of the problem) and thus it can perform effective automatic parallelisation without the need for the programmer to explicitly state where parallelism may be introduced.

NZCSRSC 2008, April 2008, Christchurch, New Zealand.

1.1 The Web Application Domain

Web applications, commonly known as "web apps", are computer programs (or groups of computer programs) that run on an Internet-connected web server. A user accesses a web server using a web browser, which connects over a network socket and communicates with the server using the HTTP protocol. All user input to the program is given at the time the request is made, in a manner that is analogous to command line parameters. The web server then invokes the requested web application with the specified parameters. A web application can be implemented in a variety of languages and development environments. Output from the program execution is sent back to the client's web browser over the same network socket. This output usually takes the form of an HTML document, which is then rendered by the web browser. No execution of code takes place on the client-side.

The fact that all execution takes place on the server-side gives developers a lot of flexibility when it comes to developing dynamically-generated web sites. Often a database will be employed to maintain data persistence between individual user requests (each invocation of the web application terminates upon completion of its output generation, removing any possibility of data persistence).

Web applications are most often written in an imperative, monolithic style. Code responsible for business logic is intermingled with display code. Separation of concerns into functions or modules is rare, and is generally only done to a limited degree. This is because many web applications are developed as "small" projects, with small budgets and very little pre-planning or regard for normal software development practice. The limited audiences for most web applications generally encourage a rapid-prototyping development approach which ignores constraints such as resources or future scalability. Unfortunately, the nearly unlimited potential audience available to a site on the web means that the popularity of a site can increase dramatically, leaving little room for poorly constructed applications to be scaled up.

One of the most popular web application development languages is PHP, often used in conjunction with the free SQL database system, MySQL [8]. The readable syntax and lack of many "advanced" programming constructs (such as threading) makes it ideal and accessible for novice programmers. It does not provide any readily-available mechanisms for separating logic and display, nor for any kind of encapsulation or modularity [1]. A common method for scaling up a web application written in PHP is to attempt crude front-end caching of results, adding read-only database instances on separate servers to attempt to ease the load on any one point in the system. In the most extreme cases, the application may be rewritten entirely. While these strategies tend to be a fairly effective in practice, the time and cost associated with scaling an application upwards in this manner raises questions about the efficacy of this solution. For example, it is unlikely that a system relying on these methods could scale upwards at the same rate as the increase in demand.

1.2 Code and Data Distribution

The introduction of concurrency can be an extremely effective means of scaling up a web application [3]. This may take the form of parallelism in the program (that is, splitting the program into pieces that may be executed on separate processors), or parallelism in the database, where the persistent data-store may be divided among multiple processors or servers. While the focus of this project is primarily on the former strategy, it is possible to achieve both by using the same mechanism for the automatic extraction of parallelism from sequential program code. The output from this automatic parallelisation process is both n parallel threads, as well as a template for the distribution of a program's data.

1.3 Related Work

The problem of extracting non-loop parallelism from general purpose languages such as FORTRAN and C has been examined at length by [6, 7]. However, the techniques they describe seek to introduce parallelism into computationally-intensive programs, in contrast to the data-intensive nature of web applications.

Many others have attempted to extract fine-grained parallelism from high-performance applications so as to increase throughput or optimise a particular aspect of computation, such as loop parallelism or vectorisation (e.g. [14, 2, 5]). These approaches are almost universally geared towards exploiting raw computational power for primarily numerical applications, such as computer graphics or scientific computing. There is generally little or no need to this type of fine-grained parallelism in web applications, where typical parallelism is employed as a means of enabling scalability, rather than attempting to optimize a particular specialised numerical task.

2. CONCURRENCY

Concurrency is a property of systems which consist of multiple components executing at the same time and possibly interacting [13]. This is a desirable property within modern applications because it enables the efficient utilisation of parallel hardware, such as multi-core processors. Similarly, by limiting the ways in which the individual components of a concurrent system may communicate (i.e. limiting communication to explicit message-passing rather than implied communication through the use of shared variables), we may also utilise parallel hardware more efficiently through the distribution of the data that a given concurrent system relies upon.

2.1 Task-level parallelism

Within the general realm of parallel execution strategies, there are various ways of characterising parallel workloads. The most common target of automatic parallelisation efforts is *loop-level parallelism*, which is usually considered to be a special case of *fine-grained parallelism*, whereby the size of each parallel work unit is relatively small. Loop-level parallelism attempts to execute independent iterations of a loop as separate "threads". The rationale for this type of parallelism is that most of the time spent executing a program is spent in loops. This type of parallelism, however, is not necessarily useful for web applications. Unlike more general computing applications, the display-intensive nature of web applications is unlikely to lend itself to small, independent loops. Instead of this loop-level parallelism, the focus of this project is statement-level *coarse-grained parallelism*. With this in mind, an individual statement is the smallest unit which will be considered for parallelisation. The resulting concurrent processes will each consist of approximately n/m statements, where n is the total number of statements in the program and m is the number of parallel threads to be constructed.

3. SEMANTIC ANALYSIS

When approaching automatic parallelisation, it is useful to consider a sequential program to be the parallel composition of two or more parallel programs, where the "true concurrency" information has been discarded (for the necessary purpose of writing the program down in some order). This is why automatic parallelisation is usually conceptualised as the "extraction" of concurrency information from a program, rather than the introduction of such information.

When considering concurrency within programs at a per-statement level, the main focus of semantic analysis is to determine which statements share *dependencies*. This is most often characterised by a case in which two statements attempt to read and/or write the same variable in some order. If statement A assigns to variable x the value 10, and statement B adds 1 to the value of x, then these two statements both share a dependency on x. The consequence of two such statements sharing a data dependency is that the order in which they are executed becomes significant. Assuming x is initialised to zero, the effect of executing A and then B is very different to executing B and then A (in the former case, x will have a final value of 11, in the latter, x will have a final value of 10). Obviously if some statement C that assigned the value 10 to y were considered, the final state of the variables x and y would remain the same no matter which order A and C were executed in. $A; C$ (A and then C in sequence), $C; A$ and indeed $A||C$ (A executed in parallel with C) are all *semantically equivalent*. For the purposes of this project, it is important at all later stages to define what it means for two programs to be semantically equivalent.

Definition: For any two programs P_1 and P_2, and an initial environment E, $P_1 \equiv P_2$ if $E \xrightarrow{P_1} F$ and $E \xrightarrow{P_2} G$ such that $F = G$.

This definition gives us a means by which to evaluate any potential program transformation step taken during parallelisation. Two programs are semantically equivalent iff, after termination, they leave the environment in the exact same state. Environmental equality in this case includes the order in which any output is generated, as well as the order in which any interactions with data that is external to the program occur.

The issue of non-terminating programs deserves some mention in this definition of equality. Within the web application domain, there are a number of elements (e.g. the client's web browser, the web server and various network connections along the way) that will "time out" waiting for a divergent web application to terminate. For this reason, we consider the post-state for a non-terminating program to simply be the state in which the environment is left once the program is (forcibly) terminated by a time-out. For the purposes of "observational equality", it is reasonable to assume that the time-out occurs at the same point during the execution of each program being observed. Furthermore, given that the exact behaviour of a non-terminating program is not well-defined for web applications, it is not a particularly important distinction within this domain.

3.1 Atomicity of control structures

Much of the focus of automatic parallelisation efforts has been on loop-level parallelism. This is a reasonably well-established technique (e.g. [14, 2, 5]). So as to focus instead on the comparatively novel *non-loop* parallelisation techniques, we treat all loops as atomic operations, rather than attempting any loop parallelisation. While there does not appear to be any reason that the paralleliser we describe could not be extended to incorporate loop parallelisation techniques, this has not yet been investigated.

In order to allow for and $while$ loops to be parallelised along with the rest of the program, they are simply treated as atomic, indivisible program statements that may be parallelised only as a whole (and not as individual loop iterations).

Similarly, attempting to parallelise program code within a compound $if-else$ block represents additional complexity within the paralleliser that serves little purpose. In order for if statements to be properly parallelised, it is assumed that a simple syntactic transformation is performed as some early stage of the compilation process, in which an if block featuring n statements is broken up into n if statements with a single statement in the body.

3.2 Aliasing

A common problem facing parallelisation schemes (particularly those concerned with loop-level parallelism) is *aliasing*. Aliasing occurs where two (or more) different variables may refer to the same region of memory, and consequently updating one variable causes the value of both variables to change. The most common examples of aliasing would be *pointers* in C, *references* in C++/Java and *variable variables* in some versions of PHP. The presence of aliasing in a program causes problems for even the most sophisticated parallelisation schemes (e.g. [11]). For our purposes, aliasing is specifically disallowed in all cases.

4. PARALLELISM EXTRACTION

In general terms, the strategy we have developed for the extraction of parallelism from sequential programs attempts to add annotations to each statement of the program that give a complete description of its behaviour with regard to program state (i.e. reading and writing regions of memory associated with variables).

Once all the statements in the program are "guarded" by annotations, it is possible to move statements into other threads in such a way that their "view" of the program state remains unchanged.

The statement annotations consists of assertions (or groups of assertions) guaranteeing certain values for what we term *sequence numbers*. These are essentially "version numbers" for variables. When some variable a is introduced within the current scope, its sequence number (denoted N_a) is initialised to zero. The set N contains sequence numbers for each and every variable in the program. Where some duplicate name may exist, all occurrences of that variable may be renamed within that scope to some unique name, without changing the meaning or intent of the program.

These *sequence numbers* closely resemble the "subscripts" used in *static single assignment form*, an intermediate representation format often employed by optimising compilers [4].

4.1 Statement Annotation

Annotations are of the form $\{P\}\ S\ \{Q\}$, as is standard within conventional Hoare logic [9]. However, while conventional Hoare logic dictates that P and Q would contain an assertion for each variable in S, the modified Hoare logic that we use for parallelism extraction instead uses assertions for the sequence number of each variable in S. It is also important to note that our modified logic (being very simple) is able to be applied forwards (that is, the post-condition Q is calculated from the pre-condition P), rather than backwards as is the case in traditional Hoare logic as it is used for program correctness proofs.

In the simplest terms, an annotation for a particular statement will require in the pre-condition that a particular "version" of a variable be available within the current environment (i.e. that a particular version of a variable is the current one) and will make a guarantee in the post-condition either that the variable may never be updated by that statement, or it will reflect the possibility of the variable being updated (whether it actually is updated or not is immaterial) by incrementing the version number for that variable.

For a simple statement such as $a = 1$, the pre- and post-conditions would take the form: $\{N_a = 0\}\ a = 1\ \{N_a = 1\}$, stating that in order for this statement to execute with its original intent, a must not have been updated by any other statement prior to this point in the execution, and after this statement is executed, a will have been updated at most once (i.e. once or potentially not at all). A sequence number with the value 0 denotes an *uninitialised* variable, that has neither been declared within the current scope, nor assigned a value. Upon the declaration of a variable (e.g. `num a = 123`), the sequence number is incremented from 0 to 1.

4.2 Definitions

To make the formalisation of the parallelisation process easier, we define a function $\rho(S)$ that takes a statement S and returns the set of variables which occur within it. For example, $\rho(a = b + c) = \{a, b, c\}$.

Similarly, to help describe the result of combining pre- and post-conditions for different statements, we define an "overwrite" operator ($\triangleright$) and a function max:

$$N_x \triangleright N_y \triangleq \begin{cases} N_x & \text{if } N_x \geq N_y \\ N_y & \text{if } N_x < N_y \end{cases}$$

$$max(Q, V) \triangleq [x | N_x \in Q, N_y \in V, x = N_x \triangleright N_y]$$

The max function takes two sets of post-conditions (Q and V) as arguments and for all assertions of the form $N_x = k$ that occur in both sets, returns only the occurrence with the highest value of k. This allows for the construction of a single post-condition for a statement (such as if) where the post-conditions of two sub-clauses could potentially each contain different conflicting sequence number values for the same variable.

Finally, the definition of what it means for a given environment E (essentially the set of values in N at a given point in time) to *satisfy* a pre-condition P must be formalised:

$$sat(P, E) \triangleq \forall N_x \in P . N_x \in E$$

This relation simply states that for each predicate in P, there is a corresponding statement in E that makes the predicate true.

4.2.1 Assignment

$$\{k > 0, N_x = k, N_1^E = v_1, ...\} x := E \{N_x = k + 1, N_1^E = v_1, ...\}$$

For assignment, the sequence number of the target variable x will always be incremented in the post-condition. The sequence numbers for all other variables in $\rho(E)$ ($N_1^E, N_2^E, ...$) remain unchanged between the pre- and post-condition. A statement such as $a = b + c$ will be annotated as:

$\{N_a = k, N_b = l, N_c = m\}\ a = b + c\ \{N_a = k + 1, N_b = l, N_c = m\}$

4.2.2 Sequential Composition

$$\frac{\{P\}S\{Q\}, \{U\}T\{V\}, U \subset Q}{\{P\}S; T\{(Q - U) \cup V\}}$$

The sequential composition of a program S and a statement T results in a program that includes T appended to S. It must be the case that the pre-condition of T is a subset of the post-condition of S in order for Q to satisfy U. This may be shown to be the case in all correct programs, as any variable in U that did not also appear in Q would be undefined, and therefore the program will be rejected by the compiler.

4.2.3 While Rule

$$\frac{\{P\}B\{Q\}}{\{P\}while\ b\ do\ B\ done\{Q\}}$$

The pre- and post-conditions for a $while$ statement are the pre- and post-conditions for the body, B, which will contain assertions about the variables that occur in the body, in addition to the elements of the boolean condition b. The number of iterations that $while$ will perform is unimportant in this case, as it is treated as *atomic*, so the pre- and post-conditions need only indicate whether or not a given variable can potentially be modified during the execution of the loop or not. The number of times which the variable is modified during loop execution is immaterial.

4.2.4 If Rule

$$\frac{\{P\}B\{Q\}, \{U\}C\{V\}}{\{P \cup U\}if\ b\ then\ B\ else\ C\{(Q \Delta V) \cup max(Q, V)\}}$$

The pre-condition for an if statement is the union of the pre-conditions of the sub-clauses, B and C, with a condition for the conditional boolean variable b. The post-condition is computed by taking the union of the symmetric difference of Q and V (all elements in one set but not in both, denoted Δ) and the result of $max(Q, V)$. This returns a post-condition equivalent to the union of both post-conditions, removing conflicting sequence number conditions.

Obviously, because the boolean condition b could potentially be undecidable at compile-time (or indeed non-terminating!), it is not possible to say which of B or C will actually be executed at runtime. To accommodate this "limitation" (which essentially equates to non-determinism), the system assumes that the result of the entire if statement will be the "worst-case", where the "worst case" is the largest sequence number that each variable could possibly be associated with after the execution of either B *or* C.

The max function is used to find the worst-case post-condition, insofar as that while B may only modify a given variable once (therefore increasing its sequence number by one), the C clause may modify the same variable n times (increasing its sequence number by n). The sequence number for that variable after evaluation of the entire if statement is taken to be the greater of the two.

4.2.5 Output

$$\frac{\{P\}E\{Q\}}{\{N_{output} = k, P\}print\ E\{N_{output} = k + 1, Q\}}$$

The pre- and post-condition for an output statement are the pre- and post-conditions for the output expression, E, in conjunction with a sequence number condition for the *output* global variable. The manner in which output ordering is preserved will be discussed more fully in subsection 6.2.

4.3 Coverage and soundness

These rules are derived directly from the grammar for the Ceth language. Through the complete application of the rules above to a well-formed Ceth program in-order, one will obtain a *guarded program* that satisfies the following conditions:

- For a given statement $\{P\}S\{Q\}$, where P and Q are computed using the rules above, $\forall x \in \rho(S).N_x \in P \wedge N_x \in Q$.
- For two statements $\{P_1\}S_1\{Q_1\}$ and $\{P_2\}S_2\{Q_2\}$, $\forall x \in (\rho(S_1) \cap \rho(S_2)).(N_x = k) \in P_1 \wedge (N_x = j) \in P_2 \wedge k \leq j$.
- For any assignment statement $\{P\}x := E\{Q\}$, $N_x \in P < N_x \in Q$.

In the simplest terms, the guarantee there will never be a "free" variable in any statement S (i.e. one that is not referenced in a pre- and post-condition), is enough to guarantee coverage of all statements.

5. EXECUTION STRATEGY

The actual parallelisation process is designed to take output from the early stages of compilation (primarily the *Abstract Syntax Tree*) to which sequence number annotations have been added, and generate an *execution strategy*. We use the term "execution strategy" to refer to the set of statement-to-sub-program mappings and the set of sub-program-to-physical node mappings, i.e. both the configuration in which statements are distributed across parallel sub-programs and the distribution of the sub-programs and their dependent datasets across parallel elements of the target execution environment (e.g. some architecture-specific notion of a "thread", a processor core or a cluster node). The most important aspect of execution strategy generation is that the resulting sub-programs, when executed in parallel, are semantically equivalent to the original input program.

The definition of semantic equivalence given in subsection 3 is very general, but it is sufficient for the purposes of ensuring that any transformation that is performed in the course of parallelising a program does not change the meaning of that program (by our definition, its *observable effect* on program state).

The first step towards constructing a parallel execution strategy within Ceth is to initialise n empty sub-programs (as specified by the programmer). Each sub-program has its own *environment*, that consists of a set of sequence variables that are particular to that sub-program. The effect of declaring a variable (and therefore incrementing its sequence number) in one sub-program is restricted to that sub-program's environment. It has no effect on the environment of any other sub-program within the execution strategy. The only way in which one sub-program can modify the environment of another is through an explicit *synchronisation*.

A synchronisation is a communication between two sub-programs, where the value of some variable is exchanged. In practice this means that one sub-program with an out-of-date copy of a variable will synchronise with some other sub-program so as to obtain the latest copy. Both sub-programs pause their execution and wait for the synchronisation to finish upon encountering a "sync" instruction. When executing the program, the effect of a synchronisation is to exchange a value, however, during the semantic analysis phase, the effect of a synchronisation is to update the sequence number for a given variable in one thread to reflect the sequence number for that variable in another thread. This process is modeled by the *Synchronisation rule:*

$$\frac{\{P\}S\{Q\} \parallel \{U\}T\{V\}}{\{k > 0, N_x = k, P\}S\{Q\} \parallel \{U\}T\{V, N_x = k\}}$$

5.1 Naïve execution strategy

The generation of an execution strategy is a reasonably straight-forward process that we have developed. The process is parameterised on the number of threads n that the user wishes to generate from a single input program S.

5.2 Technique

1. Taking the first statement $\{U\}s\{V\} \in S$, calculate the subset $C(s)$ of sub-programs to which s may potentially be added ($C(s) = \{p | p \in P \wedge sat(U, p)\}$). This is the set of sub-programs in which the current environment (corresponding to post-condition of the last statement currently in the sub-program) satisfies the pre-condition of s. Randomly select one of these sub-programs and add s to it (according to the *Sequential composition* rule given in subsection 4.1).
2. If there is no sub-program that can satisfy U, add a *synchronisation* $p_1 \xrightarrow{a} p_2$ where $p1, p2 \in P \wedge p_1 \neq p_2 \wedge a \in \rho(s)$.
3. Repeat steps 1 and 2 until all statements in S have been placed within a sub-program.

Program 1 An example program

```
num apples = random();
num oranges = random();

if(apples < oranges) print "We need more apples!";

if(apples > oranges) print "We need more or-
anges!";
```

The generation of a suitable execution strategy for Program 1 begins with the addition of annotations, to create the so-called *guarded program* (Program 2).

From this guarded program, it is possible to begin the distribution of statements across N sub-programs. For the purposes of this example we will assume $N = 2$, however, it should be reasonably apparent that this technique generalises to cases where $N > 2$.

Considering each statement in-order, it is necessary to first place statement 1. Its precondition requires that $N_{apples} =$

Program 2 A guarded program

$\{N_{apples} = 0\}$ `num apples = random();` $\{N_{apples} = 1\}$
$\{N_{oranges} = 0\}$ `num oranges = random();` $\{N_{oranges} = 1\}$

$\{N_{apples} = 1, N_{oranges} = 1, N_{output} = 0\}$
`if(apples < oranges) print "We need more apples!";`
$\{N_{apples} = 1, N_{oranges} = 1, N_{output} = 1\}$

$\{N_{apples} = 1, N_{oranges} = 1, N_{output} = 1\}$
`if(apples > oranges) print "We need more oranges!";`
$\{N_{apples} = 1, N_{oranges} = 1, N_{output} = 2\}$

0. This is satisfied immediately, as all sequence numbers are assumed to be initialised to 0 within the "empty" environment by default. Therefore we may (randomly) choose to place statement 1 in sub-program 1 or sub-program 2. Assuming we choose sub-program 1, the environment for sub-program 1 is updated to reflect the addition of the statement. The set of sequence numbers is updated so that $N_{apples} = 1$ within sub-program 1, while remaining unchanged for sub-program 2.

Continuing in this manner, assume statement 2 is placed randomly in in sub-program 2. In order to place statement 3, we require a sub-program environment in which $N_{apples} = 1$ and $N_{oranges} = 1$. Since no such sub-program exists, it is necessary to add a synchronisation. Choosing the variable *apples* from sub-program 1 as the synchronisation *source* and sub-program 2 as the synchronisation *target*, it is now possible to place statement 3 in sub-program 2, as the environment now satisfies the pre-condition. Table 1 shows the actions taken in each step in order to construct the eventual execution strategy.

This process is not the only means of constructing an execution strategy, however, it does have the advantage of being very simple, and very efficient in terms of time and space.

In the two sub-programs derived from Table 1, there is a degree of asymmetry present in the arrangement of statements. The size of this example is such that we have actually increased the total number of statements (adding one synchronisation). Over larger programs though, the decomposition of statements is intended to result in the division of many independent elements with only a few synchronisations added.

6. SOUNDNESS

THEOREM 1. *At each point in the construction of the execution strategy using the technique above, the pre-conditions of all statements placed within sub-programs are satisfied by the environment in which they are placed, i.e., for any sub-program S consisting of $\{P\}S_0; S_1; ...; S_{n-1}\{Q\}; \{U\}S_n\{V\}$, it is always the case that $sat(U, Q)$ is true.*

PROOF. *By induction on the structure of the sub-program. Assume we have two possibly-empty sub-programs, $\{P_A\}A\{Q_A\}$ and $\{P_B\}B\{Q_B\}$, for which the condition above is true, and an annotated source program S.*

Base Case: The empty pre-condition (which the very first statement of S must have, as it may not require the existence of anything within the empty environment) is satisfied by the newly-initialised environment, i.e., $sat(\{\}, \phi) = true$.

Now we must consider two cases:

Case: *If $C(S_n) \neq \phi$, then $sat(S_n, Q_A)$ or $sat(S_n, Q_B)$ is true, by the definition of $C(S_n)$. In this case, the* sequential composition rule *may be applied to S_n and $\{P_x\}x\{Q_x\}$ (where $x \in C(S_n)$), which yields a sub-program for which the sat condition above is true.*

Case: *If $C(S_n) = \phi$, then S_n may not be composed with either sub-program without the addition of at least one synchronisation. A finite number of synchronisations will yield a sub-program which satisfies S_n for three reasons:*

1. *$\rho(P) \subseteq \rho(P; sync(x))$ (a synchronisation will always introduce zero or one new variables into the environment of P)*
2. *$\rho(P) \subseteq \rho(S)$ (i.e. there are never more variables in the environment of a sub-program than there are in the original program, as none of the rules given in subsection 4.1 provide a means of introducing bindings in P that are not present in S).*
3. *$\rho(s) \subseteq \rho(S)$ where $s \in S$ (i.e. there are never more variables in the environment of a given statement than there are in the entire program).*

By the definition of sat, we can see that any sub-program environment E that contains bindings for all the elements of $\rho(S_n)$ may be composed with S_n such that $sat(S_n, E)$ is true. □

This case with 2 sub-programs may be shown to extend to n sub-programs by induction over the number of sub-programs.

THEOREM 2. *Having established correctness of the construction process, we now show that the resulting sub-programs share the same pre- and post-conditions as the original input program, i.e.:*

Taking a program $\{P\}S\{Q\}$ and two sub-programs, $\{P_1\}S_1\{Q_1\}$ and $\{P_2\}S_2\{Q_2\}$, $P \rightarrow P_1 \cup P_2$ and $Q \rightarrow Q_1 \cup Q_2$, allowing us to conclude that $\{P\}S\{Q\} \equiv \{P_1 \cup P_2\}S_1 \parallel S_2\{Q_1 \cup Q_2\}$

PROOF. *By induction on the structure of the sub-programs, S_1 and S_2.*

Base Case: If P is empty, P_1 and P_2 are also empty, as the execution strategy generation steps provide no means of introducing new bindings in P_1 or P_2 that are not present in P (i.e. $P_1 \cup P_2 \subseteq P$). Similarly, if P is empty, Q is empty, as Q cannot contain statements about unguarded variables ($P \rightarrow Q$).

Case: *Given that each statement in S is in $S_1 \cup S_2$, and no other statements are in $S_1 \cup S_2$, the pre- and post-conditions are identical (as no rules are provided to manipulate assertions during execution strategy generation).*

S	$C(S)$	SP1 Env.	SP2 Env.	Action
1	$\{1,2\}$	$\{\}$	$\{\}$	Add to SP1
2	$\{1,2\}$	$\{N_{apples}=1\}$	$\{\}$	Add to SP2
3	ϕ	$\{N_{apples}=1\}$	$\{N_{oranges}=1\}$	SP1 $\xrightarrow{apples}$ SP2
3	$\{2\}$	$\{N_{apples}=1\}$	$\{N_{oranges}=1, N_{apples}=1\}$	Add to SP2
4	$\{2\}$	$\{N_{apples}=1\}$	$\{N_{oranges}=1, N_{apples}=1, N_{output}=1\}$	Add to SP2
-	-	$\{N_{apples}=1\}$	$\{N_{oranges}=1, N_{apples}=1, N_{output}=2\}$	-

Table 1: Execution strategy generation steps

By inspection, one can see that $P_1 \cup P_2 = P$ *and* $Q_1 \cup Q_2 = Q$*, therefore* $P \rightarrow P_1 \cup P_2$ *and* $Q \rightarrow Q_1 \cup Q_2$. □

6.1 Optimisation

The execution strategy resulting from the initial construction process is likely to be far from optimal, encompassing many more synchronisations than are actually required, and possibly resulting in a wildly uneven distribution across the sub-programs. In order to search for a globally optimal solution, a *Simulated Annealing* (SA) optimisation technique [10] is employed. To use this technique, it must be possible to compute a numerical score that indicates the optimality of a particular solution (an *energy function*, in SA parlance). The energy function that we use in Ceth is given as Eq. 1.

$$energy(P,\alpha,\beta) = \alpha \sum_{i=0}^{|P|} (|P_i| - \overline{|P|})^2 + \beta \left(\sum_{j=0}^{|P|} |P_j| - s \right)^{-1} \quad (1)$$

This energy function for some execution strategy P is tuned according to two weighting parameters; the importance of an even distribution of program code and data across n sub-programs (α), or minimisation of the number of synchronisations that have been added in order to generate a solution (β). These tuning properties allow for the execution strategy to reflect the hardware realities of the target execution environment. In shared-memory systems with fast inter-process communication, the cost of a synchronisation may be very low, in which case it makes sense to maximise the number of sub-programs. In other situations, the cost of a synchronisation may be large, in which case a programmer may choose to sacrifice a larger number of sub-programs in favour of minimising the number of synchronisations. The ability to create a new execution strategy with a simple re-compilation is an extremely valuable aspect of this technique. This could allow system administrators to handle unexpected increases in the use of a web application by simply adding new hardware, without requiring any modification of the program code.

6.2 Output ordering

One of the most visible outcomes from the non-determinism arising from parallel execution of programs that generate output is that without special consideration, the order in which output is generated may be unpredictable and inconsistent. Consider a program (and equivalent three threads) that is supposed to generate "1", "2" and "3" as output. Since the program consists of three separate *print* statements, the paralleliser places each print statement in a separate thread. The parallel execution of these three threads may generate output in any order, resulting in outputs such as 321, 213, 132 etc., depending on the exact order in which program execution reaches the *print* statement in each thread. In order to guarantee that output is generated in the same order by the parallelised version of the program as in the original, it is necessary to provide a mechanism for the compiler to guarantee this property.

Output statements such as "print" are annotated using the *Output rule*. This rule makes reference to an implicit parameter *output*. The effect of using a print statement is the same as assigning to the output variable, modifying its sequence number. The output variable is defined by the compiler as existing only in sub-program 1. Through optimisation, it is usually the case that print statements end up in sub-program 1 (avoiding extra synchronisations). Through the sequence number mechanism, it is possible to maintain output ordering as written in the original program without having to restrict output in any particular way.

6.3 Data distribution

So far the discussion of automatic parallelisation has primarily concerned the distribution of program statements across sub-programs, however, there is another aspect to achieving improved scalability and performance through parallelisation - Data distribution, whereby persistent data external to the program itself is stored on separate servers. Within this system, the way in which data is accessed and modified is guarded by sequence numbers in the same way that the semantics of statements are preserved. While persistent data in web applications is most commonly stored in relational databases, it is possible to model the manner in which they are used much more simply. For the purposes of distributing data across execution nodes, we use a very conservative strategy, choosing to not divide datasets in cases where it would introduce extra complexity despite being "correct" in some sense. The operations we expose for operating on persistent data are as follows:

- $Create(Type, Key, Value)$ - Create an object of type $Type$, indexed by Key that holds the value $Value$.
- $Read(Key, Type)$ - Retrieve the value currently indexed by Key.
- $Update(Key, Type, Value)$ - Replace the current value indexed by Key with $Value$.
- $Delete(Key, Type)$ - Delete the entry indexed by Key.
- $Map(Fn, Type)$ - For every object o of type $Type$, replace its current value with the result of calling the function $Fn(o)$.

These operations should be reasonably familiar to anyone who has used relational databases. If one conceptualises "objects" as simple containers that are approximately equivalent to rows in a database table, and each distinct object "type" as a different database table, we quickly arrive at a set of primitive operations with reasonably trivial mappings to most SQL database operations used in web applications. For example, *Create* is equivalent to an SQL $INSERT$, while *Read* corresponds to a single-item $SELECT$ by key. *Update* and *Delete* map directly to their SQL counter-parts. The *Map* operation can model a $SELECT$ or $UPDATE$ of multiple items in a database table. The conservative nature of this approach becomes clear when we consider a simple SQL statement such as $UPDATE\ table1\ SET\ cost = 1\ WHERE\ cost > 10$. If we suppose that *table*1 has 100 rows, 10 of which have a value greater than 10 in the *cost* column, a total of 10 rows would be updated upon executing this SQL command. Within the Ceth data model, however, the "type" would be *table*1, and a *Map* command will be used with some function that checks for a *cost* value greater than 10 and returns a modified object if the condition is true. Because the *Map* command may potentially modify every object (or "row") of type *table*1, in the worst-case, every member of *table*1 has been modified.

We have chosen to annotate statements with sequence numbers on a *per-type* basis, where the modification of any one object of a given type causes the sequence number for that type to be incremented. In modeling an SQL "JOIN", where multiple tables are accessed or modified, the sequence numbers for all types involved are updated. In the case of reading and accessing objects, the pre-condition demands that the sequence number for the type be some value k.

The inclusion of persistent data in the annotations for individual statements provides the ability for data to be partitioned in the same way that statements that depend upon it can be. If a given type has no sequence number within the environment of a given sub-program, it is assumed to be inaccessible from within that environment, and therefore statements within that sub-program must synchronise with another sub-program to access that data (in which case the energy function used in optimisation dictates that it is more optimal for statements that depend on particular data to be closer to the data). While it would likely be undesirable for Ceth to perform the data partitioning automatically, it outputs enough information at compile-time to allow a programmer to confidently partition data in such a way that the program will run as though it were still using a single database.

7. FUTURE WORK

The most obvious additional feature for this paralleliser would be incorporating loop-level parallelisation within the same mechanism, allowing for the extraction of two types of parallelism using only one set of annotations. The combination of two types of parallelism could provide greater benefits for more computationally-intensive applications. Similarly, the techniques employed here could easily be extended to other program domains, with potential to be refined as a general technique that may be applicable to any imperative language that does not allow for aliasing (or which allows aliasing to be reliably factored out).

8. CONCLUSION

Automatic parallelisation is increasingly relevant given recent trends in hardware configurations, and given the prevalence of web applications, work within this domain seems to be of some significance. We have demonstrated that a system such as Ceth can be used to successfully extract task-level parallelism from sequential web applications using the techniques we have developed.

9. REFERENCES

[1] M. Achour, F. Betz, A. Dovgal, N. Lopes, P. Olson, G. Richter, D. Seguy, and J. Vrana. *PHP Manual*. The PHP Documentation Group, 2007.

[2] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, et al. Effective Automatic Parallelization with Polaris. *International Journal of Parallel Programming, May*, 1995.

[3] S. Ceri. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, 2002.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[5] A. Eichenberger, J. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, et al. Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[6] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):166–178, 1992.

[7] M. Girkar and C. Polychronopoulos. Extracting task-level parallelism. *ACM transactions on programming languages and systems*, 17(4):600–634, 1995.

[8] H. Hartman. Tools for dynamic Web sites: ASP vs PHP vs ASP .NET. *Seybold Report Analysing Publishing Technologies*, 1:12, 2001.

[9] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] S. Kirkpatrick, C. Gelatt Jr, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671, 1983.

[11] Z. Li and P. Yew. Program parallelization with interprocedural analysis. *The Journal of Supercomputing*, 2(2):225–244, 1988.

[12] N. Mukherjee and J. Gurd. A comparative analysis of four parallelisation schemes. *Proceedings of the 13th international conference on Supercomputing*, pages 278–285, 1999.

[13] A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, London ; New York, 1998.

[14] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.