Project Proposal: Incorporating Better Register Allocation into Jikes

Jonathan Derryberry
Manfred Lau

October 26, 2003

- **Group Info:** Jonathan Derryberry (jonderry@cs.cmu.edu), Manfred Lau (mlau@cs.cmu.edu).
- **Project Web Page:** http://www-2.cs.cmu.edu/~jonderry/15745project/project.htm
- **Project Description:** Currently the Jikes VM uses two compilers to perform just in time compilation of java bytecodes. One compiler is a fast compiler, which is used on each method the first time it is called. This compiler is very fast but performs few, if any, optimizations. When code is executed many times, thus becoming "hot," a second compiler is invoked, which performs simple optimizations on the code. Register allocation is not performed at all in the first compiler, but in the second compiler a simple linear time register allocation algorithm is executed. The reason why a more complicated algorithm is not used is that, for Jikes, compilation is occurring at runtime, so it is essential that the register allocation not spend too much time devising an allocation that only yields a marginal improvement in performance. In fact, the linear time register allocation algorithm used in Jikes typically achieves an allocation that is within a small factor of optimal.

  In our project, we will attempt to outfit Jikes with a more sophisticated register allocation scheme. Instead of using a binary hot versus cold heuristic for whether to improve the performance of the code, we will attempt to achieve finer grained control over optimization. More formally, let $T(0)$ denote the execution time of a method before any register allocation has been performed, and let $T(t)$ denote the execution time of a method after t units of time have been spent searching for the best register allocation scheme. Generally, $T(t)$ is expected to be a decaying function that is asymptotic to T_{min}, the execution time of the method given optimal register allocation. On the other side, let $N(n)$ be the expected number of times that a method expects to be called given that it has already been called n times. It is reasonable to assume that $N(n) = n$, which satisfies our intuition that if one method has been called twice as many times as another, that method is expected to be called twice as many times as the other before the application terminates, and our intuition that applications that have been running for a long time are expected to continue to run for a long time, on average. The value of $N(n)$ is closely related to the marginal benefit of running the register allocation method for another short period of time. In particular, the marginal effect of improving register allocation on the remaining execution time of the program is given by

$$ME(t) = -\frac{dT(t)}{dt} \cdot N(n) + \frac{dt}{dt} = -\frac{dT(t)}{dt} \cdot N(n) + 1. \quad (1)$$

  Intuitively, Equation 1 says that performing a small amount of work (dt) on the register allocation problem yields a benefit of $\frac{dT(t)}{dt}$ units of time for each time the method is executed, and because the method is expected to execute $N(n)$ more times, we expect the effect of a

small amount of work on the remaining run time of the program to be $-\frac{dT(t)}{dt} \cdot N(n)$. However, to measure the net effect of the additional work on runtime, we must add the time we spend computing the register allocation.

We will be trying to incorporate these theoretical ideas into Jikes, to see if we can improve runtime performance. In practice, we could define $T(t)$ to be the value of the heuristic for determining the spill cost of a particular allocation (e.g. penalize each spill by a factor of 10^d, where d is the nesting depth). After performing the optimizations for an interval of time, the change in $T(t)$ could be measured, and used to approximate the expected benefit b from executing the allocation for another interval of time. Then, a counter C could be initialized to compute when $ME(t)$ would become negative (net time savings); C would be decremented each time the method is called and when it hit 0, another phase of register allocation would commence. For example, the algorithm could find a suitable C by solving $-\frac{bn}{\tau} + 1 < \alpha$ for n, where τ is the amount of time spent in each step of the register allocation algorithm and α is a threshold for when the next phase of register allocation is deemed sufficiently profitable to merit execution.

The success of this technique could be measured by running several benchmark programs and measuring the impact of using the new register allocation algorithm on execution time as compared to the original algorithm.

- **Logistics:**
  - **Plan of Attack and Schedule:**
    1. **10/27:** Get Jikes running and familiarize ourself with how the current register allocator works.
    2. **11/03:** Devise (or look up) a suitable "anytime" register allocation algorithm that is capable of the type of marginal improvement of its solution indicated above.
    3. **11/10:** Get an initial version of the register allocation algorithm operational within Jikes.
    4. **11/17:** Debug initial version of the algorithm.
    5. **11/24:** Search for inevitable necessary improvements in the heuristics in order to exceed the performance of the initial algorithm, while simultaneously testing the comparative execution time performance.
    6. **12/01:** Continue debugging, perform final testing, do final writeup, expect spillover of other tasks into this week.

    The critical path in this schedule is likely to be the process of getting a working register allocator to work with the Jikes VM, because we have no experience with Jikes. Work is expected all to be done together.
  - **Milestone:** Have an operational, bug-free register allocator working within the Jikes VM.
  - **Literature:** The Jikes Research Virtual Machine website: http://www-124.ibm.com/developerworks/oss/jikesrvm/, "The Jalapeno dynamic optimizing compiler for Java" (M. Burke et. al.), "Linear scan register allocation" (Massimiliano Poletto and Vivek Sarkar), "Adaptive optimization in the Jalapeño JVM" (Matthew Arnold et. al.).
  - **Resources Needed:** We only need Jikes, which is freely available, so we have all that we need.
  - **Getting Started:** We have spoken with Pedro Artigas to get background information on Jikes. We are having difficulty getting Jikes running, but otherwise we are all set to begin.