Агрессивная инлайн-подстановка функций для VLIW-архитектур

А.В. Ермолицкий <era@mcst.ru>
М.И. Нейман-заде <muradnz@mcst.ru>
О.А. Четверина <chetverina_o@mcst.ru>
А.Л. Маркин <markin_a@mcst.ru>
В.Ю.Волконский <vol@mcst.ru>
АО "МЦСТ", 119991, Россия, г. Москва, Ленинский проспект, дом 51

Аннотация. Достижение высокой производительности на микропроцессорах с VLIW-архитектурой возможно лишь при использовании агрессивной инлайн-подстановки. Предложенный в настоящей работе алгоритм оптимизации явно учитывает время компиляции, что делает его эвристику более сбалансированной и позволяет значительно сократить рост кода и ускорить компиляцию по сравнению с известными алгоритмами. Кроме того, нам удалось достичь высоких показателей производительности благодаря ряду факторов: учёт в эвристике ключевых оптимизаций, использование клонирования функций, частичной инлайн-подстановки и компиляции в режиме «вся программа». Реализация нашего алгоритма в оптимизирующем компиляторе для архитектуры Эльбрус позволила ускорить задачи SPEC CPU2006 в среднем в 1.41 раз.

Ключевые слова: оптимизация, оптимизирующий компилятор, инлайн-подстановка, VLIW.

DOI: 10.15514/ISPRAS-2015-27(6)-13

Для цитирования: Ермолицкий А.В., Нейман-заде М.И., Четверина О.А., Маркин А.Л., Волконский В.Ю. Агрессивная инлайн-подстановка функций для VLIW-архитектур. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 189-198. DOI: 10.15514/ISPRAS-2015-27(6)-13.

1. Введение

Большинство современных программ написано с использованием высокоуровневых языков программирования. При этом распространённой практикой является разбиение исходного кода на множество мелких функций. Это упрощает разработку, однако препятствует эффективному исполнению программы, поскольку код различных функций не может перемешиваться на этапе компиляции. Широко известным решением указанной проблемы

является инлайн-подстановка (inline expansion), выполняемая оптимизирующим компилятором [1-5]. Суть данной оптимизации заключается в подстановке тела функции в точку её вызова. Наряду с очевидным устранением операций вызова, передачи параметров и результата, инлайн-подстановка позволяет увеличить эффективность других оптимизаций, таких как наложение итераций цикла и планирование инструкций. В частности, инструкции из разных процедур могут быть спланированы одновременно. Основным недостатком инлайн-подстановки является рост размера кода, что приводит к увеличению времени компиляции. Чрезмерный рост кода приводит к возникновению блокировок из-за более частых промахов в кэш кода. Кроме того, в некоторых случаях перемешивание холодного и горячего кода может приводить к неоптимальному планированию последнего.

Указанные проблемы особенно актуальны для VLIW-архитектур со статическим планированием, где в случае небольших функций сложно использовать возможности параллельного исполнения инструкций. Для VLIW необходима очень агрессивная инлайн-подстановка [6], которая в случае современных программ может выполнять подстановку десятков тысяч вызовов. Известные алгоритмы инлайн-подстановки приводят к слишком сильному росту кода и замедлению компиляции. Как правило, авторы подобных алгоритмов решают задачу о ранце (Knapsack problem), т.е. пытаются найти такое множество подстановок, при котором время исполнения программы минимально, а коэффициент увеличения размера кода не превышает заданной величины. Такая постановка задачи на практике нередко приводит к избыточному росту кода, либо алгоритм останавливается до достижения существенного прироста производительности. Частичная подстановка функций [7-12] решает указанные проблемы лишь отчасти, в общем случае её область применимости сильно ограничена. Кроме того, эти алгоритмы показывают хорошие результаты лишь при наличии профильной информации и компиляции в режиме "вся программа" (whole program), на практике такие режимы компиляции используются исключительно редко.

2. Сбалансированная инлайн-подстановка

В настоящей работе представлен алгоритм агрессивной инлайн-подстановки, в котором за счёт тщательного подбора эвристик удалось достичь оптимального баланса между уменьшением времени исполнения *Te* и увеличением времени компиляции *Tc*. Эвристика алгоритма основана на минимизации функционала: $Te^3Tc \rightarrow min$ (1). За счёт учёта *Tc*, предложенный алгоритм обладает хорошей масштабируемостью, позволяя оптимизировать в режиме whole огромные программы, содержащие миллионы строк исходного кода. В теории такая эвристика не ограничивает рост времени компиляции, однако на практике она позволяет значительно уменьшить в среднем *Tc* по сравнению с известными алгоритмами агрессивной инлайн-подстановки.

В общем случае задача минимизации функционала (1) является NP-полной [13]. Мы используем жадный алгоритм для приближённого решения этой задачи, суть его заключается в следующем. Формируется множество всех операций вызова в программе, для которых возможно выполнение инлайн-подстановки (*call_set*). Для каждого вызова из *call_set* вычисляется его вес: $W = -3\Delta Te/Te - \Delta Tc/Tc$, где ΔTe и ΔTc - изменение времени исполнения и компиляции программы соответственно, вызванное данной инлайн-подстановкой. Подстановка считается полезной только при $W > 0$. Инлайн-подстановки выполняются в порядке уменьшения веса до тех пор, пока есть хоть один вызов с $W > 0$. При этом в процессе работы оптимизации для подставленных вызовов вычисляется вес и они добавляются в множество *call_set*. Кроме того, учитывается, что подстановка одной функции может изменить веса других функций - поддерживается динамическое перевычисление весов и переупорядочивание множества вызовов.

Основной сложностью при реализации эвристики является оценка величин *Te* и *Tc*. Для оценки *Te* каждая функция разбивается на регионы - циклы и линейные участки. Для каждого региона оценивается время его исполнения в зависимости от количества инструкций в нём, и затем эти времена суммируются. При этом используется профильная информация: статическая (на основе оценки компилятором вероятностей переходов) либо динамическая (полученная в результате тренировочного запуска программы). Для корректной оценки *Te* и ΔTe необходимо, чтобы профильная информация была согласованной, т.е. сумма счётчиков вызовов функции должна равняться счётчику стартового узла этой функции. На практике статический профиль часто бывает несогласованным, это происходит в первую очередь из-за наличия рекурсии и вызовов по указателю. Поэтому перед оценкой *Te* выполняется локальное согласование профиля за счёт умножения счётчиков функции на корректирующий множитель. Время компиляции оценивается по формуле: $Tc = a + bN^2$, где *a* и *b* - константы, подобранные эмпирически, а *N* - количество инструкций в функции. Подобная оценка *Tc* является очень грубой, поскольку реальное время компиляции зависит от множества факторов, однако для наших целей такое приближение оказалось достаточным.

Ключевой особенностью алгоритма является учёт различных оптимизаций, которые становятся возможными в результате инлайн-подстановки, делается это следующим образом. При оценке веса вызова $f \rightarrow g$ вычисляется изменение времени исполнения: $\Delta Te = (Te(f^*) + Te(g^*)) - (Te(f) + Te(g))$, где $Te(f^*)$ и $Te(g^*)$ - оценка времени исполнения функций *f* и *g* соответственно после инлайн-подстановки функции *g* в *f*. Оптимизации учитываются при вычислении $Te(f^*)$. Например, если функция *g* вызывается с константными параметрами, учитывается, что после подстановки g результат некоторых выражений можно будет вычислить статически, некоторые ветвления управления можно будет удалить, соответственно, количество подставленных

в *f* инструкций будет меньше, чем изначально было в *g*. Таким образом, при оценке $Te(f^*)$ мы учитываем только те инструкции, которые останутся после инлайн-подстановки и выполнения оптимизаций. Аналогичным образом учитываются оптимизации при оценке $Tc(f^*)$ и ΔTc. Мы учитываем следующие оптимизации:

- планирование кода (code scheduling)
- пропагация значений (global copy propagation)
- пропагация констант (constant propagation)
- удаление избыточных ветвлений управления
- удаление мёртвого кода (dead code elimination)
- наложение итераций цикла (softpipe)
- прочие цикловые оптимизации [14,15].

Кроме того, вес дополнительно увеличивается в случае, когда в результате подстановки уточняется количество итераций цикла (если оно задаётся параметром функции) либо удаётся порвать больше зависимостей между указателями - параметрами подставляемой функции.

3. Экспериментальные результаты

Предложенный алгоритм был внедрён в промышленный оптимизирующий компилятор для архитектуры Эльбрус [16,17]. Для эмпирического подбора коэффициентов в функциях оценки *Te* и *Tc* использовались задачи из пакета SPEC CPU2000 [18]. Коэффициенты подбирались таким образом, чтобы минимизировать среднеквадратичное отклонение оценочных и реальных величин *Te* и *Tc*. Для проверки эффективности предложенного алгоритма использовались 29 задач из пакета SPEC CPU2006 [18]. Замеры времени исполнения задач проводились на четырёхъядерном микропроцессоре Эльбрус 4С с тактовой частотой 800 МГц. В помодульном режиме сборки без динамической профильной информации (опции -O3 -ffast) за счёт инлайн-подстановки время исполнения в среднем уменьшилось на 29%, при этом время компиляции увеличилось всего на 13%, а размер бинарного файла увеличился на 9.1% (см. таблицу 1). В режиме "вся программа" с использованием профиля (опции -O3 –ffast –fwhole –fprofile-use) результаты получились ещё более впечатляющими: исполнение ускорилось на 41%, компиляция замедлилась на 12%, размер бинарного файла увеличился на 7.7% в среднем (см. таблицу 2). Наибольшее ускорение было достигнуто на задаче 447.dealII, которая ускорилась в 7.3 раза при незначительном замедлении компиляции (0.9%) и уменьшении размера кода (1.5%).

Таблица 1: результаты для помодульного режима компиляции

название теста	коэффициент ускорения теста	замедление компиляции	рост размера бинарного кода
400.perlbench	1,00	7,9%	7,3%
401.bzip2	1,00	0%	4,0%
403.gcc	1,03	11,4%	8,3%
410.bwaves	1,00	3,9%	0%
416.gamess	1,01	5,5%	3,1%
429.mcf	1,09	6,0%	19,2%
433.milc	1,00	21,4%	11,3%
434.zeusmp	4,39	17,3%	-1,4%
435.gromacs	1,07	14,5%	13,9%
436.cactusADM	1,00	7,3%	5,2%
437.leslie3d	1,03	1,4%	0,2%
444.namd	2,43	44,4%	25,0%
445.gobmk	1,02	10,9%	4,8%
447.dealII	4,32	46,6%	34,0%
450.soplex	1,41	10,0%	4,7%
453.povray	1,70	31,6%	30,6%
454.calculix	1,00	0%	1,5%
456.hmmer	1,00	7,0%	11,5%
458.sjeng	1,04	10,5%	11,3%
459.GemsFDTD	1,00	0%	0%
462.libquantum	1,03	14,5%	10,4%
464.h264ref	1,01	2,7%	4,8%
465.tonto	1,01	3,1%	5,4%
470.lbm	1,00	2,3%	5,7%
471.omnetpp	1,56	15,5%	10,7%
473.astar	1,43	20,2%	10,3%
481.wrf	1,52	2,3%	2,6%
482.sphinx3	1,00	9,1%	8,9%
483.xalancbmk	3,27	11,1%	20,1%
gmean	**1,29**	**13,0%**	**9,1%**

Таблица 2: результаты для режима компиляции «вся программа»

название теста	коэффициент ускорения теста	замедление компиляции	рост размера бинарного кода
400.perlbench	1,03	13,9%	12,9%
401.bzip2	1,00	8,0%	5,7%
403.gcc	1,10	21,2%	22,2%
410.bwaves	1,00	0%	0%
416.gamess	1,02	0%	0%
429.mcf	1,28	0%	-1,7%
433.milc	1,88	41,8%	23,9%
434.zeusmp	4,38	9,4%	-7,3%
435.gromacs	1,06	8,2%	7,8%
436.cactusADM	1,00	2,0%	2,8%
437.leslie3d	1,11	0%	1,4%
444.namd	2,43	50,7%	48,3%
445.gobmk	1,16	10,8%	7,3%
447.dealII	7,30	0,9%	-1,5%
450.soplex	1,52	20,0%	6,3%
453.povray	2,32	23,3%	23,1%
454.calculix	1,01	10,3%	8,9%
456.hmmer	1,01	7,2%	3,0%
458.sjeng	1,10	16,3%	14,8%
459.GemsFDTD	1,00	1,9%	0,5%
462.libquantum	1,03	3,9%	7,7%
464.h264ref	1,01	3,5%	2,7%
465.tonto	1,03	2,3%	2,3%
470.lbm	1,00	-2,1%	0,7%
471.omnetpp	1,81	23,7%	21,6%
473.astar	1,74	23,3%	7,1%
481.wrf	1,62	1,8%	0,8%
482.sphinx3	1,06	9,3%	10,2%
483.xalancbmk	3,27	-1,8%	7,7%
gmean	**1,41**	**11,8%**	**7,7%**

Список литературы

[1]. Robert W. Scheifler. An analysis of inline substitution for a structured programming language. Communications of the ACM, Sept. 1977, Volume 20, Issue 9, p. 647-654

[2]. P. C. Chang, Wen-mei W. Hwu. Inline function expansion for compiling C programs. ACM SIGPLAN Notices, 1989, vol. 24, no. 7, p. 246-257

[3]. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. Software - Practice & Experience, May 1992, Volume 22, Issue 5, p. 349 - 369

[4]. Matthew Arnold, Stephen J. Fink, Vivek Sarkar, Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. ACM SIGPLAN Notices, 2000, vol. 35, no. 7, p. 52-64

[5]. Peng Zhao, Jose Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. Workshop on Languages and Compilers for Parallel Computing (LCPC) , Oct. 2003, p. 405-419

[6]. Andrew Ayers, Richard Schooler, Robert Gottlieb. Aggressive inlining. ACM SIGPLAN Notices, 1997, vol. 32, no. 5, p. 134-145

[7]. Robert Muth, Saumya Debray. Partial Inlining. Technical report, Department of Computer Science, University of Arizona, 1997

[8]. Tom Way, Ben Breech, Lori Pollock. Region Formation Analysis with Demand-Driven Inlining for Region-Based Optimization. Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, 2000

[9]. Tom Way, Lori L. Pollock. A Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, Volume 2, p. 552-556

[10]. Dibyendu Das. Function inlining versus function cloning. ACM SIGPLAN Notices, June 2003, Volume 38, Issue 6, p. 23 - 29

[11]. Peng Zhao, Jose Nelson Amaral. Function outlining and partial inlining. Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, 2005, p. 101 - 108

[12]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim. Aggressive Function Splitting for Partial Inlining. Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, 2011, p. 80-86

[13]. M.R. Garey, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York.1979. 338 p.

[14]. Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, San Francisco. 1997. 888 p.

[15]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, Boston. 2006. 1000 p.

[16]. Краткое описание архитектуры Эльбрус, http://www.elbrus.ru/arhitektura_elbrus

[17]. В.Ю. Волконский, А.В.Брегер, А.Ю.Бучнев, А.В.Грабежной, А.В.Ермолицкий, Л.Е.Муханов, М.И.Нейман-заде, П.А.Степанов, О.А.Четверина. Методы распараллеливания программ в оптимизирующем компиляторе. Вопросы радиоэлектроники, серия ЭВТ, выпуск 3, 2012, стр.63-88

[18]. Standard Performance Evaluation Corporation. SPEC CPU Benchmarks, http://www.spec.org/

Aggressive Inlining for VLIW

A.Ermolitckii <era@mcst.ru>
M.Neiman-Zade <muradnz@mcst.ru>
O.Chetverina <chetverina_o@mcst.ru>
A.Markin <markin_a@mcst.ru>
V.Volkonskii <vol@mcst.ru>
MCST, 51 Leninskii avenue, Moscow, 119991, Russian Federation

Abstract. Inline expansion is very important for high performance VLIW, especially for microprocessors with static scheduling. Optimizations in optimizing compilers for VLIW duplicate code aggressively and lead to long compile time. Our inlining algorithm is based on heuristics that takes into account compile time explicitly. This made optimization more balanced and significantly reduced code growth and compile time compared to common inlining approach based on minimization of runtime within constraints. Instead of using hard constraints we are trading run time for compilation time in some proportion. Our heuristics predicts several key optimizations in evaluation of runtime and compile time: code scheduling, global copy propagation, dead code elimination and different loop optimizations. Optimizations prediction reduces the need in profile information which is rarely available in practice. Our implementation of inlining includes cloning, partial inlining and inlining across compilation modules in whole program mode. All this factors make dramatic impact on performance: our inlining implementation in the Elbrus optimizing compiler boost SPEC CPU2006 benchmark performance by factor of 1.41 at the cost of 12% increase of compile time and 7.7% increase of code size on average.

Keywords: optimization, optimizing compiler, inline expansion, VLIW.

DOI: 10.15514/ISPRAS-2015-27(6)-13

For citation: Ermolitckii A., Neiman-Zade M., Chetverina O., Markin A., Volkonskii V. Aggressive Inlining for VLIW. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 189-198 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-13

References

[1]. Robert W. Scheifler. An analysis of inline substitution for a structured programming language. Communications of the ACM, Sept. 1977, Volume 20, Issue 9, p. 647-654

[2]. P. C. Chang, Wen-mei W. Hwu. Inline function expansion for compiling C programs. ACM SIGPLAN Notices, 1989, vol. 24, no. 7, p. 246-257

[3]. Pohua P. Chang, Scott A. Mahlke, William Y. Chen, Wen-mei W. Hwu. Profile-guided automatic inline expansion for C programs. Software - Practice & Experience, May 1992, Volume 22, Issue 5, p. 349 - 369

[4]. Matthew Arnold, Stephen J. Fink, Vivek Sarkar, Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. ACM SIGPLAN Notices, 2000, vol. 35, no. 7, p. 52-64

[5]. Peng Zhao, Jose Nelson Amaral. To inline or not to inline? Enhanced inlining decisions. Workshop on Languages and Compilers for Parallel Computing (LCPC) , Oct. 2003, p. 405-419
[6]. Andrew Ayers, Richard Schooler, Robert Gottlieb. Aggressive inlining. ACM SIGPLAN Notices, 1997, vol. 32, no. 5, p. 134-145
[7]. Robert Muth, Saumya Debray. Partial Inlining. Technical report, Department of Computer Science, University of Arizona, 1997
[8]. Tom Way, Ben Breech, Lori Pollock. Region Formation Analysis with Demand-Driven Inlining for Region-Based Optimization. Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, 2000
[9]. Tom Way, Lori L. Pollock. A Region-based Partial Inlining Algorithm for an ILP Optimizing Compiler. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, 2002, Volume 2, p. 552-556
[10]. Dibyendu Das. Function inlining versus function cloning. ACM SIGPLAN Notices, June 2003, Volume 38, Issue 6, p. 23 - 29
[11]. Peng Zhao, Jose Nelson Amaral. Function outlining and partial inlining. Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, 2005, p. 101 - 108
[12]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim. Aggressive Function Splitting for Partial Inlining. Proceedings of the 2011 15th Workshop on Interaction between Compilers and Computer Architectures, 2011, p. 80-86
[13]. M.R. Garey, D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York.1979. 338 p.
[14]. Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers, San Francisco. 1997. 888 p.
[15]. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Pearson Education, Boston. 2006. 1000 p.
[16]. Kratkoe opisanie arkhitektury Elbrus [Short description of Elbrus architecture], http://www.elbrus.ru/arhitektura_elbrus (in Russian)
[17]. V.Volkonskii, A.Breger, A.Buchnev, A.Grabezhnoi, A.Ermolitckii, L.Mukhanov, M.Neiman-Zade, P.Stepanov, O.Chetverina. Metody rasparallelivaniia program v optimiziruiushchem kompiliatore [Program parallelization methods in optimizing compiler]. Voprosy radioelektroniki, seriia EVT, vypusk 3 [Questions of radioelectronics, Computer Technology series, Volume 3], 2012, p.63-88 (In Russian)
[18]. Standard Performance Evaluation Corporation. SPEC CPU Benchmarks, http://www.spec.org/