An LLVM-based Hybrid Binary Translation System

* 論文中的HBT作法是一般狀況都使用SBT，當遇到run-time exception再改用DBT，而轉換的過程中用LLVM做轉碼、最佳化、產生目的碼，概念上是將執行檔的二元碼轉成IR，再從IR轉成目標ISA的二元碼
* Static binary translation(SBT) :

可以對整個程式做最佳化且不增加程式的執行時間，但當遇到indirect branch時會很難處理，如果二元碼是被某些特定的編譯器像是GCC產生的話，SBT可以解決indirect branch的問題，但當有手寫的組合語言包含indirect branch時，SBT很難處理

* Dynamic binary translation(DBT) :

可以輕鬆處理indirect branch的問題，但會增加執行時間，做最佳化的效果也沒有SBT好，也會增加memory的使用，雖然這些問題在普遍的計算環境下沒有太大的問題，但在embedded computing上卻不可容許。現今的binary translation都使用DBT，因為它能有效地解決code discovery及code location的問題，

* LLVM :

是一個開源的編譯器框架，能將機器獨立的指令轉成機器相依的組合語言，它包含了一個just-in-time的編譯器及Machine Code toolkit，toolkit中有跟CPU指令相關的工具，像是assembler, disassembler, object file format handling，在近幾年快速發展後，增加了多種的進階最佳化功能及分析能力

* 要解決的問題 :

SBT在手寫的indirect branch指令要如何處理

如何取得放在code segment的資料

* HBT中的SBT處理過程

1. Source binary經由Object reader切成多段的data
2. Data經由LLVMMC disassembler轉成LLVM MC IR (bit code file)
3. Static translator將LLVM MC IR轉成LLVM IR (轉為machine-independent的類似組合語言的中間語言)
4. LLVM optimizer做LLVM IR的target-independent最佳化
5. LLVM static compiler 將上面的bit code轉成目標的assembly，再經由system assembler轉成object files
6. System linker將object files, source image, linker script, run-time system(包含dynamic translator, system call emulator)連接在一起得到target binary

* HBT中的DBT使用時間點及處理過程

1. 做初始化，建立一個dynamic translator、分配stack空間給轉換後的binary、將引數放到source stack中
2. 從program entry point開始執行
3. 執行中若遇到indirect branch，則去address mapping table中找在target binary對應的address
4. 若address mapping table沒有對應的address，則切換到dynamic translator並查詢indirect branch的destination有沒有在code cache中，若在就直接執行，不在dynamic translator就去source image將含有destination的code做轉換，並放到code cache中在開始執行

* HBT中的dynamic translation process

1. Dynamic translator持續地將連續指令block轉成LLVM IR，直到遇到branch指令，並對IR做基本的最佳化
2. LLVM IR經由LLVM JIT compiler轉成native code，並執行對特定目標的最佳化
3. 將block entry的source address及它對應的native code的entry存到address translation table中供code cache查詢用

* 結論 : HBT比一般的DBT平均上快8倍